DWARF Debugging Information Format

Version 4

DWARF Debugging Information Format Committee

http://www.dwarfstd.org

June 10, 2010

DWARF Debugging Information Format, Version 4

This document is based in part on the DWARF Debugging Information Format, Version 2, which contained the following notice:

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

This document is further based on the DWARF Debugging Information Format, Version 3, which is subject to the GNU Free Documentation License.

Trademarks:

Table of Contents

DWARF DEBUGGING INFORMATION FORMAT VERSION 4 I

1 INTRODUCTION 1

1.1 PURPOSE AND SCOPE 1
1.2 OVERVIEW 1
1.3 VENDOR EXTENSIBILITY 2
1.4 CHANGES FROM VERSION 3 TO VERSION 4 3
1.5 CHANGES FROM VERSION 2 TO VERSION 3 4
1.6 CHANGES FROM VERSION 1 TO VERSION 2 5

2 GENERAL DESCRIPTION 7

2.1 THE DEBUGGING INFORMATION ENTRY (DIE) 7
2.2 ATTRIBUTE TYPES 7
2.3 RELATIONSHIP OF DEBUGGING INFORMATION ENTRIES 16
2.4 TARGET ADDRESSES 16
2.5 DWARF EXPRESSIONS 17
2.5.1 General Operations 17
2.5.2 Example Stack Operations 25
2.6 LOCATION DESCRIPTIONS 25
2.6.1 Single Location Descriptions 26
2.6.2 Location Lists 30
2.7 TYPES OF PROGRAM ENTITIES 32
2.8 ACCESSIBILITY OF DECLARATIONS 32
2.9 VISIBILITY OF DECLARATIONS 33
2.10 VIRTUALITY OF DECLARATIONS 33
2.11 ARTIFICIAL ENTRIES 34
2.12 SEGMENTED ADDRESSES 34
2.13 NON-DEFINING DECLARATIONS AND COMPLETIONS 35
2.13.1 Non-Defining Declarations 35
2.13.2 Declarations Completing Non-Defining Declarations 36
2.14 DECLARATION COORDINATES 36
2.15 IDENTIFIER NAMES 36
2.16 DATA LOCATIONS AND DWARF PROCEDURES 37
2.17 CODE ADDRESSES AND RANGES 37
2.17.1 Single Address 38
2.17.2 Contiguous Address Range 38
2.17.3 Non-Contiguous Address Ranges 38
2.18 ENTRY ADDRESS 40
2.19 STATIC AND DYNAMIC VALUES OF ATTRIBUTES 40
2.20 ENTITY DESCRIPTIONS 41
2.21 BYTE AND BIT SIZES 41
2.22 LINKAGE NAMES 41

3 PROGRAM SCOPE ENTRIES 43

3.1 UNIT ENTRIES 43

3.1.1 Normal and Partial Compilation Unit Entries *43*
3.1.2 Imported Unit Entries *47*
3.1.3 Separate Type Unit Entries *48*
3.2 MODULE, NAMESPACE AND IMPORTING ENTRIES 48
3.2.1 Module Entries *49*
3.2.2 Namespace Entries *49*
3.2.3 Imported (or Renamed) Declaration Entries *50*
3.2.4 Imported Module Entries *51*
3.3 SUBROUTINE AND ENTRY POINT ENTRIES 53
3.3.1 General Subroutine and Entry Point Information *53*
3.3.2 Subroutine and Entry Point Return Types *55*
3.3.3 Subroutine and Entry Point Locations *55*
3.3.4 Declarations Owned by Subroutines and Entry Points *55*
3.3.5 Low-Level Information *56*
3.3.6 Types Thrown by Exceptions *57*
3.3.7 Function Template Instantiations *57*
3.3.8 Inlinable and Inlined Subroutines *58*
3.3.9 Trampolines *64*
3.4 LEXICAL BLOCK ENTRIES 65
3.5 LABEL ENTRIES 65
3.6 WITH STATEMENT ENTRIES 66
3.7 TRY AND CATCH BLOCK ENTRIES 66

4 DATA OBJECT AND OBJECT LIST ENTRIES 69

4.1 DATA OBJECT ENTRIES 69
4.2 COMMON BLOCK ENTRIES 73
4.3 NAMELIST ENTRIES 73

5 TYPE ENTRIES 75

5.1 BASE TYPE ENTRIES 75
5.2 UNSPECIFIED TYPE ENTRIES 80
5.3 TYPEDEF ENTRIES 82
5.4 ARRAY TYPE ENTRIES 83
5.5 STRUCTURE, UNION, CLASS AND INTERFACE TYPE ENTRIES 84
5.5.1 Structure, Union and Class Type Entries *84*
5.5.2 Interface Type Entries *86*
5.5.3 Derived or Extended Structs, Classes and Interfaces *86*
5.5.4 Access Declarations *87*
5.5.5 Friends *87*
5.5.6 Data Member Entries *88*
5.5.7 Member Function Entries *92*
5.5.8 Class Template Instantiations *93*
5.5.9 Variant Entries *94*
5.6 CONDITION ENTRIES 95
5.7 ENUMERATION TYPE ENTRIES 96
5.8 SUBROUTINE TYPE ENTRIES 97
5.9 STRING TYPE ENTRIES 98
5.10 SET TYPE ENTRIES 98
5.11 SUBRANGE TYPE ENTRIES 99

5.12 Pointer to Member Type Entries 100
5.13 File Type Entries 101
5.14 Dynamic Type Properties 102
5.14.1 Data Location 102
5.14.2 Allocation and Association Status 102
5.15 Template Alias Entries 103

6 OTHER DEBUGGING INFORMATION 105

6.1 Accelerated Access 105
6.1.1 Lookup by Name 106
6.1.2 Lookup by Address 107
6.2 Line Number Information 108
6.2.1 Definitions 109
6.2.2 State Machine Registers 109
6.2.3 Line Number Program Instructions 111
6.2.4 The Line Number Program Header 112
6.2.5 The Line Number Program 115
6.3 Macro Information 123
6.3.1 Macinfo Types 123
6.3.2 Base Source Entries 125
6.3.3 Macinfo Entries for Command Line Options 125
6.3.4 General Rules and Restrictions 125
6.4 Call Frame Information 126
6.4.1 Structure of Call Frame Information 127
6.4.2 Call Frame Instructions 131
6.4.3 Call Frame Instruction Usage 136
6.4.4 Call Frame Calling Address 137

7 DATA REPRESENTATION 139

7.1 Vendor Extensibility 139
7.2 Reserved Values 140
7.2.1 Error Values 140
7.2.2 Initial Length Values 140
7.3 Executable Objects and Shared Objects 140
7.4 32-Bit and 64-Bit DWARF Formats 140
7.5 Format of Debugging Information 143
7.5.1 Unit Headers 143
7.5.2 Debugging Information Entry 145
7.5.3 Abbreviations Tables 145
7.5.4 Attribute Encodings 146
7.6 Variable Length Data 161
7.7 DWARF Expressions and Location Descriptions 163
7.7.1 DWARF Expressions 163
7.7.2 Location Descriptions 167
7.7.3 Location Lists 167
7.8 Base Type Attribute Encodings 168
7.9 Accessibility Codes 170
7.10 Visibility Codes 171
7.11 Virtuality Codes 171

7.12 Source Languages 171
7.13 Address Class Encodings 173
7.14 Identifier Case 174
7.15 Calling Convention Encodings 174
7.16 Inline Codes 175
7.17 Array Ordering 175
7.18 Discriminant Lists 176
7.19 Name Lookup Tables 176
7.20 Address Range Table 177
7.21 Line Number Information 178
7.22 Macro Information 180
7.23 Call Frame Information 180
7.24 Non-contiguous Address Ranges 182
7.25 Dependencies and Constraints 183
7.26 Integer Representation Names 184
7.27 Type Signature Computation 184

APPENDIX A -- ATTRIBUTES BY TAG VALUE (INFORMATIVE) 191

APPENDIX B -- DEBUG SECTION RELATIONSHIPS (INFORMATIVE) 213

APPENDIX C -- VARIABLE LENGTH DATA: ENCODING/DECODING (INFORMATIVE) 217

APPENDIX D -- EXAMPLES (INFORMATIVE) 219

D.1 Compilation Units and Abbreviations Table Example 219
D.2 Aggregate Examples 221
D.2.1 Fortran 90 Example 221
D.2.2 Ada Example 227
D.2.3 Pascal Example 230
D.3 Namespace Example 232
D.4 Member Function Example 235
D.5 Line Number Program Example 237
D.6 Call Frame Information Example 239
D.7 Inlining Examples 243
D.7.1 Alternative #1: inline both OUTER and INNER 244
D.7.2 Alternative #2: Inline OUTER, multiple INNERs 247
D.7.3 Alternative #3: inline OUTER, one normal INNER 250
D.8 Constant Expression Example 252
D.9 Unicode Character Example 254
D.10 Type-Safe Enumeration Example 255
D.11 Template Example 256
D.12 Template Alias Examples 259

APPENDIX E -- DWARF COMPRESSION AND DUPLICATE ELIMINATION (INFORMATIVE) 263

E.1 Using Compilation Units 263
E.1.1 Overview 263
E.1.2 Naming and Usage Considerations 265
E.1.3 Examples 269

E.2 USING TYPE UNITS 276
E.2.1 SIGNATURE COMPUTATION EXAMPLE 277
E.2.2 TYPE SIGNATURE COMPUTATION GRAMMAR 285
E.3 SUMMARY OF COMPRESSION TECHNIQUES 287
E.3.1 #INCLUDE COMPRESSION 287
E.3.2 ELIMINATING FUNCTION DUPLICATION 287
E.3.3 SINGLE-FUNCTION-PER-DWARF-COMPILATION-UNIT 287
E.3.4 INLINING AND OUT-OF-LINE-INSTANCES 288
E.3.5 SEPARATE TYPE UNITS 288
APPENDIX F – DWARF SECTION VERSION NUMBERS (INFORMATIVE) 289

List of Figures

FIGURE 1. TAG NAMES 8
FIGURE 2. ATTRIBUTE NAMES 14
FIGURE 3. CLASSES OF ATTRIBUTE VALUE 15
FIGURE 4. ACCESSIBILITY CODES 32
FIGURE 5. VISIBILITY CODES 33
FIGURE 6. VIRTUALITY CODES 33
FIGURE 7. EXAMPLE ADDRESS CLASS CODES 35
FIGURE 8. LANGUAGE NAMES 45
FIGURE 9. IDENTIFIER CASE CODES 46
FIGURE 10. CALLING CONVENTION CODES 54
FIGURE 11. INLINE CODES 59
FIGURE 12. ENDIANITY ATTRIBUTE VALUES 72
FIGURE 13. ENCODING ATTRIBUTE VALUES 77
FIGURE 14. DECIMAL SIGN ATTRIBUTE VALUES 80
FIGURE 15. TYPE MODIFIER TAGS 82
FIGURE 16. ARRAY ORDERING 83
FIGURE 17. DISCRIMINANT DESCRIPTOR VALUES 95
FIGURE 18. TAG ENCODINGS 154
FIGURE 19. CHILD DETERMINATION ENCODINGS 154
FIGURE 20. ATTRIBUTE ENCODINGS 159
FIGURE 21. ATTRIBUTE FORM ENCODINGS 161
FIGURE 22. EXAMPLES OF UNSIGNED LEB128 ENCODINGS 162
FIGURE 23. EXAMPLES OF SIGNED LEB128 ENCODINGS 163
FIGURE 24. DWARF OPERATION ENCODINGS 167
FIGURE 25. BASE TYPE ENCODING VALUES 169
FIGURE 26. DECIMAL SIGN ENCODINGS 169
FIGURE 27. ENDIANITY ENCODINGS 170
FIGURE 28. ACCESSIBILITY ENCODINGS 170
FIGURE 29. VISIBILITY ENCODINGS 171
FIGURE 30. VIRTUALITY ENCODINGS 171
FIGURE 31. LANGUAGE ENCODINGS 173
FIGURE 32. IDENTIFIER CASE ENCODINGS 174
FIGURE 33. CALLING CONVENTION ENCODINGS 174
FIGURE 34. INLINE ENCODINGS 175
FIGURE 35. ORDERING ENCODINGS 175
FIGURE 36. DISCRIMINANT DESCRIPTOR ENCODINGS 176
FIGURE 37. LINE NUMBER STANDARD OPCODE ENCODINGS 179
FIGURE 38. LINE NUMBER EXTENDED OPCODE ENCODINGS 179
FIGURE 39. MACINFO TYPE ENCODINGS 180
FIGURE 40. CALL FRAME INSTRUCTION ENCODINGS 182
FIGURE 41. INTEGER REPRESENTATION NAMES 184
FIGURE 42. ATTRIBUTES BY TAG VALUE 211
FIGURE 43. DEBUG SECTION RELATIONSHIPS 214
FIGURE 44. ALGORITHM TO ENCODE AN UNSIGNED INTEGER 217
FIGURE 45. ALGORITHM TO ENCODE A SIGNED INTEGER 217
FIGURE 46. ALGORITHM TO DECODE AN UNSIGNED LEB128 NUMBER 218

FIGURE 47. ALGORITHM TO DECODE A SIGNED LEB128 NUMBER....218
FIGURE 48. COMPILATION UNITS AND ABBREVIATIONS TABLE....220
FIGURE 49. FORTRAN 90 EXAMPLE: SOURCE FRAGMENT....221
FIGURE 50. FORTRAN 90 EXAMPLE: DESCRIPTOR REPRESENTATION....222
FIGURE 51. FORTRAN 90 EXAMPLE: DWARF DESCRIPTION....225
FIGURE 52. ADA EXAMPLE: SOURCE FRAGMENT....227
FIGURE 53. ADA EXAMPLE: DWARF DESCRIPTION....229
FIGURE 54. PACKED RECORD EXAMPLE: SOURCE FRAGMENT....230
FIGURE 55. PACKED RECORD EXAMPLE: DWARF DESCRIPTION....231
FIGURE 56. NAMESPACE EXAMPLE: SOURCE FRAGMENT....232
FIGURE 57. NAMESPACE EXAMPLE: DWARF DESCRIPTION....234
FIGURE 58. MEMBER FUNCTION EXAMPLE: SOURCE FRAGMENT....235
FIGURE 59. MEMBER FUNCTION EXAMPLE: DWARF DESCRIPTION....236
FIGURE 60. LINE NUMBER PROGRAM EXAMPLE: MACHINE CODE....237
FIGURE 61. LINE NUMBER PROGRAM EXAMPLE: ONE ENCODING....238
FIGURE 62. LINE NUMBER PROGRAM EXAMPLE: ALTERNATE ENCODING....238
FIGURE 63. CALL FRAME INFORMATION EXAMPLE: MACHINE CODE FRAGMENTS....240
FIGURE 64. CALL FRAME INFORMATION EXAMPLE: CONCEPTUAL MATRIX....241
FIGURE 65. CALL FRAME INFORMATION EXAMPLE: COMMON INFORMATION ENTRY ENCODING....241
FIGURE 66. CALL FRAME INFORMATION EXAMPLE: FRAME DESCRIPTION ENTRY ENCODING....242
FIGURE 67. INLINING EXAMPLES: PSEUDO-SOURCE FRAGMENT....243
FIGURE 68. INLINING EXAMPLE #1: ABSTRACT INSTANCE....245
FIGURE 69. INLINING EXAMPLE #1: CONCRETE INSTANCE....246
FIGURE 70. INLINING EXAMPLE #2: ABSTRACT INSTANCE....248
FIGURE 71. INLINING EXAMPLE #2: CONCRETE INSTANCE....249
FIGURE 72. INLINING EXAMPLE #3: ABSTRACT INSTANCE....251
FIGURE 73. INLINING EXAMPLE #3: CONCRETE INSTANCE....252
FIGURE 74. CONSTANT EXPRESSIONS: C++ SOURCE....252
FIGURE 75. CONSTANT EXPRESSIONS: DWARF DESCRIPTION....254
FIGURE 76. UNICODE CHARACTER TYPE EXAMPLES....254
FIGURE 77. C++ TYPE-SAFE ENUMERATION EXAMPLE....255
FIGURE 78. C++ TEMPLATE EXAMPLE #1....256
FIGURE 79. C++ TEMPLATE EXAMPLE #2....257
FIGURE 80. TEMPLATE ALIAS EXAMPLE #1....259
FIGURE 81. TEMPLATE ALIAS EXAMPLE #2....260
FIGURE 82. DUPLICATE ELIMINATION EXAMPLE #1: C++ SOURCE....269
FIGURE 83. DUPLICATE ELIMINATION EXAMPLE #1: DWARF SECTION GROUP....270
FIGURE 84. DUPLICATE ELIMINATION EXAMPLE #1: PRIMARY COMPILATION UNIT....271
FIGURE 85. DUPLICATE ELIMINATION EXAMPLE #2: FORTRAN SOURCE....272
FIGURE 86. DUPLICATE ELIMINATION EXAMPLE #2: DWARF SECTION GROUP....273
FIGURE 87. DUPLICATE ELIMINATION EXAMPLE #2: PRIMARY UNIT....274
FIGURE 88. DUPLICATE ELIMINATION EXAMPLE #2: COMPANION SOURCE....274
FIGURE 89. DUPLICATE ELIMINATION EXAMPLE #2: COMPANION DWARF....275
FIGURE 90. TYPE SIGNATURE EXAMPLES: C++ SOURCE....277
FIGURE 91. TYPE SIGNATURE COMPUTATION #1: DWARF REPRESENTATION....278
FIGURE 92. TYPE SIGNATURE COMPUTATION #1: FLATTENED BYTE STREAM....279
FIGURE 93. TYPE SIGNATURE COMPUTATION #2: DWARF REPRESENTATION....281
FIGURE 94. TYPE SIGNATURE EXAMPLE #2: FLATTENED BYTE STREAM....284
FIGURE 95. TYPE SIGNATURE EXAMPLE USAGE....285

FIGURE 96. TYPE SIGNATURE COMPUTATION GRAMMAR....286
FIGURE 97. SECTION VERSION NUMBERS....289

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), a 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 DWARF Version 2 or Version 3 to later versions should be straightforward and easily accomplished. Almost all DWARF Version 2 and Version 3 constructs have been retained unchanged in DWARF Version 4.

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 Debugging Information Format, Version 4

DWARF Committee members contributing to Version 4 are:

Todd Allen	Concurrent Computer
David Anderson	
John Bishop	Intel
Jim Blandy	CodeSourcery
Ron Brender, Editor	
Andrew Cagney	
Siu Chi Chan	IBM
Cary Coutant	Google
John DelSignore	TotalView
Michael Eager, Chair	Eager Consulting
Ben Elliston	IBM
Mike Gleeson	Hewlett-Packard
Matthew Gretton-Dann	ARM
David Gross	Hewlett-Packard
Tommy Hoffner	IBM
Jason Molenda	Apple
David Moore	Intel
Jeff Nelson	Hewlett-Packard
Chris Quenelle	Sun Microsystems
Paul Robinson	Hewlett-Packard
Bill White	TotalView
Kendrick Wong	IBM

For further information about DWARF or the DWARF Committee, see http://www.dwarfstd.org.

This document is intended to be usable in online as well as traditional paper forms. In the online form, blue text is used to indicate hyperlinks which facilitate moving around in the document in a manner like that typically found in web browsers. 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 list of all attributes which in turn contains hyperlinks for the multiple usages. The Table of Contents also provides hyperlinks to the respective sections.

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

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.

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 4, the fourth generation of debugging information based on the DWARF format. DWARF Version 4 extends DWARF Version 3 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 Sections 2 through 6. Section 2 describes the overall structure of the information and attributes that is common to many or all of the different debugging information entries. Sections 3, 4 and 5 describe the specific debugging information entries and how they communicate the necessary information about the source program to a debugger. Section 6 describes debugging information contained outside of the debugging information entries. The encoding of the DWARF information is presented in Section 7.

This organization closely follows that used in the DWARF Version 3 document. Except where needed to incorporate new material or to correct errors, the DWARF Version 3 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 Vendor Extensibility

This document does not attempt to cover all interesting languages or even to cover all of the interesting debugging information needs for its primary target languages. 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.

DWARF Version 4 is intended to be permissive rather than prescriptive. 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 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 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.

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 table sections include a additional fields that affect the location and interpretation of other data in the section.

1.5 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).
- References that use the attribute form DW_FORM_ref_addr are specified to be four bytes in the DWARF 32-bit format and eight bytes in the DWARF 64-bit format, while DWARF Version 2 specifies that such references have the same size as an address on the target system (see Sections 7.4 and 7.5.4).
- 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).

1.6 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.

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 Figure 1. The debugging information entries they identify are described in Sections 3, 4 and 5.

The debugging information entry descriptions in Sections 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 `.debug_types` sections of an object file.

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 Figure 2.

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

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

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

Figure 1. Tag names

Figure 2, Attribute names, begins here.

Attribute	Identifies or Specifies
DW_AT_abstract_origin	Inline instances of inline subprograms Out-of-line instances of inline subprograms
DW_AT_accessibility	C++ and Ada declarations C++ base classes C++ inherited members
DW_AT_address_class	Pointer or reference types Subroutine or subroutine type
DW_AT_allocated	Allocation status of types
DW_AT_artificial	Objects or types that are not actually declared in the source
DW_AT_associated	Association status of types
DW_AT_base_types	Primitive data types of compilation unit
DW_AT_binary_scale	Binary scale factor for fixed-point type
DW_AT_bit_offset	Base type bit location Data member bit location
DW_AT_bit_size	Base type bit size Data member bit size
DW_AT_bit_stride	Array element stride (of array type) Subrange stride (dimension of array type) Enumeration stride (dimension of array type)
DW_AT_byte_size	Data object or data type size
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_column	Column position of inlined subroutine call
DW_AT_call_file	File containing inlined subroutine call

Attribute	Identifies or Specifies
DW_AT_call_line	Line number of inlined subroutine call
DW_AT_calling_convention	Subprogram calling convention
DW_AT_common_reference	Common block usage
DW_AT_comp_dir	Compilation directory
DW_AT_const_value	Constant object Enumeration literal value Template value parameter
DW_AT_const_expr	Compile-time constant object Compile-time constant function
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_default_value	Default value of parameter
DW_AT_description	Artificial name or description

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_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 module initialization Entry address of subprogram Entry address of inlined subprogram
DW_AT_enum_class	Type safe enumeration definition
DW_AT_explicit	Explicit property of member function
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

Attribute	Identifies or Specifies
DW_AT_inline	Abstract instance Inlined subroutine
DW_AT_is_optional	Optional parameter
DW_AT_language	Programming language
DW_AT_linkage_name	Object file linkage name of an entity
DW_AT_location	Data object location
DW_AT_low_pc	Code address or range of addresses
DW_AT_lower_bound	Lower bound of subrange
DW_AT_macro_info	Macro information (#define, #undef)
DW_AT_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_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

Attribute	Identifies or Specifies
DW_AT_recursive	Recursive property of a subroutine
DW_AT_return_addr	Subroutine return address save location
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	Object declaration Type 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_threads_scaled	UPC array bound `THREADS` scale factor
DW_AT_trampoline	Target subroutine
DW_AT_type	Type of declaration Type of subroutine return
DW_AT_upper_bound	Upper bound of subrange
DW_AT_use_location	Member location for pointer to member type
DW_AT_use_UTF8	Compilation unit uses UTF-8 strings
DW_AT_variable_parameter	Non-constant parameter flag
DW_AT_virtuality	Virtuality indication Virtuality of base class Virtuality of function

Attribute	Identifies or Specifies
DW_AT_visibility	Visibility of declaration
DW_AT_vtable_elem_location	Virtual function vtable slot

Figure 2. Attribute names

Attribute Class	General Use and Encoding
address	Refers to some location in the address space of the described program.
block	An arbitrary number of uninterpreted bytes of data.
constant	One, two, four or eight bytes of uninterpreted data, or data encoded in the variable length format known as LEB128 (see Section 7.6.). *Most constant values are integers of one kind or another (codes, offsets, counts, and so on); these are sometimes called "integer constants" for emphasis.*
exprloc	A DWARF expression or location description.
flag	A small constant that indicates the presence or absence of an attribute.
lineptr	Refers to a location in the DWARF section that holds line number information.
loclistptr	Refers to a location in the DWARF section that holds location lists, which describe objects whose location can change during their lifetime.
macptr	Refers to a location in the DWARF section that holds macro definition information.
rangelistptr	Refers to a location in the DWARF section that holds non-contiguous address ranges.
reference	Refers to one of the debugging information entries that describe the program. There are three 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 a 64-bit signature for that type.
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.

Figure 3. Classes of attribute value

2.3 Relationship of Debugging Information Entries

A variety of needs can be met by permitting a single debugging information entry to "own" an arbitrary number of other debugging entries and by permitting the same debugging information entry to be one of many owned by another debugging information entry. This makes it possible, 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 objects.

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

While the ownership relation of the debugging information entries is represented as a tree, other relations among the entries exist, for example, a 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). 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

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.

Architectures which have multiple instruction sets are supported by the `isa` *entry in the line number information (see Section 6.2.2).*

2.5 DWARF Expressions

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

All DWARF operations are encoded as a stream of opcodes that are each followed by zero or more literal operands. The number of operands is determined by the opcode.

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

2.5.1 General Operations

Each general operation represents a postfix operation on a simple stack machine. Each element of the stack is the size of an address on the target machine. The value on the top of the stack after "executing" the DWARF expression is taken to be the result (the address of the object, the value of the array bound, the length of a dynamic string, the desired value itself, and so on).

2.5.1.1 Literal Encodings

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

2.5.1.2 Register Based Addressing

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

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

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

3. **DW_OP_bregx**
   The DW_OP_bregx operation has two operands: a register which is specified by an unsigned LEB128 number, followed by a signed LEB128 offset.

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.

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

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

3. **DW_OP_pick**
   The single operand of the DW_OP_pick operation provides a 1-byte index. A copy of the stack entry 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 becomes the second stack entry, and the second entry becomes the top of the stack.

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

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

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

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

10. **DW_OP_xderef_size**
    The DW_OP_xderef_size operation behaves like the DW_OP_xderef operation.The entry at the top of the stack is treated as an address. The second stack entry is treated as an "address space identifier" for those architectures that support multiple address spaces. The top two stack elements are popped, 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.

11. **DW_OP_push_object_address**
    The DW_OP_push_object_address operation pushes the address of the object currently being evaluated as part of evaluation of a user presented expression. This object may correspond to an independent variable described by its own 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.

12. **DW_OP_form_tls_address**
    The DW_OP_form_tls_address operation pops a value from the stack, translates it into an address in the current thread's thread-local storage block, and pushes the address. If the DWARF expression containing the DW_OP_form_tls_address operation belongs to the main executable's DWARF info, the operation uses the main executable's thread-local storage block; if the expression belongs to a shared library's DWARF info, then it uses that shared library's thread-local storage block.

    Some implementations of C and C++ support a `__thread` 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` variables declared in the main executable, and a separate block for the variables declared in each shared library. 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. DW_OP_form_tls_address leaves the computation to the consumer.

13. **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).

    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.

2.5.1.4 Arithmetic and Logical Operations

The following provide arithmetic and logical operations. Except as otherwise specified, the arithmetic operations perfom addressing arithmetic, that is, unsigned arithmetic that is performed modulo one plus the largest representable address (for example, 0x100000000 when the size of an address is 32 bits). Such 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 and pushes the result.

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

12. **DW_OP_shl**
    The DW_OP_shl operation pops the top two stack entries, shifts the former second entry left (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,
   - 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.

   Comparisons are performed as signed operations. The six operators are DW_OP_le (less than or equal to), DW_OP_ge (greater than or equal to), DW_OP_eq (equal to), DW_OP_lt (less than), DW_OP_gt (greater than) and DW_OP_ne (not equal to).

2. **DW_OP_skip**
   DW_OP_skip is an unconditional branch. Its single operand is a 2-byte signed integer constant. The 2-byte constant is the number of bytes of the DWARF expression to skip 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 subroutine 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). The operand is used as the offset of a debugging information entry in a `.debug_info` or `.debug_types` section which may be contained in a shared object or executable other than that containing the operator. For references from one shared object or executable to another, the relocation must be performed by the consumer.

   Operand interpretation of DW_OP_call2, DW_OP_call4 and DW_OP_call_ref is exactly like that for DW_FORM_ref2, DW_FORM_ref4 and DW_FORM_ref_addr, respectively (see Section 7.5.4).

   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 Special Operations

There is one special operation 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.

2.5.2 Example Stack Operations

The stack operations defined in Section 2.5.1.3 are fairly conventional, but the following examples illustrate their behavior graphically.

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

2.6 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 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.

   Single location descriptions are of two kinds:

   a. Simple location descriptions, which describe the location of one contiguous piece (usually all) of an object. A simple location description may describe a location in addressable memory, or in a register, or the lack of a location (with or without a known value).

   b. Composite location descriptions, which describe an object in terms of pieces each of which may be contained in part of a register or stored in a memory location unrelated to other pieces.

2. *Location lists,* which are used to describe objects that have a limited lifetime or change their location during their lifetime. Location lists are more completely described below.

The two forms are distinguished in a context sensitive manner. As the value of an attribute, a location description is encoded using class exprloc and a location list is encoded using class loclistptr (which serves as an offset into a separate location list table).

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 (as though on its own separate stack, if any).

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 Memory Location Descriptions

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

2.6.1.1.2 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 ("Register Based Addressing") 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 name a register.

Note that the register number represents a DWARF specific mapping of numbers onto the actual registers of a given architecture. The mapping should be chosen to gain optimal density and should be shared by all users of a given architecture. It is 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 (see Section 2.5.1.2).

2.6.1.1.3 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 block representing the value in the memory representation of the target machine. The length operand gives the length in bytes of the block.

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.

2.6.1.1.4 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).

2.6.1.2 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 kind of 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.

2.6.1.3 Example Single Location Descriptions

Here are some examples of how DWARF operations are 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.
```

2.6.2 Location Lists

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

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

A location list entry consists of:

1. A beginning address offset. This address offset has the size of an address and is relative to the applicable base address of the compilation unit referencing this location list. It marks the beginning of the address range over which the location is valid.

2. An ending address offset. This address offset again has the size of an address and is relative to the applicable base address of the compilation unit referencing this location list. It marks the first address past the end of the address range over which the location is valid. The ending address must be greater than or equal to the beginning address.

A location list entry (but not a base address selection or end of list entry) whose beginning and ending addresses are equal has no effect because the size of the range covered by such an entry is zero.

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

The applicable base address of a location list entry is determined by the closest preceding base address selection entry (see below) in the same location list. If there is no such selection entry, then the applicable base address defaults to the base address of the compilation unit (see Section 3.1.1).

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

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

A base address selection entry consists of:

1. The value of the largest representable address offset (for example, 0xffffffff when the size of an address is 32 bits).

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

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

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

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

A base address selection entry and an end of list entry for a location list are identical to a base address selection entry and end of list entry, respectively, for a range list (see Section 2.17.3) in interpretation and representation.

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 Section 5 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 Figure 4.

DW_ACCESS_public
DW_ACCESS_private
DW_ACCESS_protected

Figure 4. Accessibility codes

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 Figure 5.

DW_VIS_local
DW_VIS_exported
DW_VIS_qualified

Figure 5. Visibility codes

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 Figure 6.

DW_VIRTUALITY_none
DW_VIRTUALITY_virtual
DW_VIRTUALITY_pure_virtual

Figure 6. Virtuality codes

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:

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

Figure 7. Example address class codes

2.13 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 variable x rather than of the local version.

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.

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.

It is not the case that 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 clearly 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). 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 is a string representing the name as it appears in the source program. A debugging information entry containing no name attribute, or containing a name attribute whose value consists of a name containing a single null byte, represents a program entity for which no name was given in the source.

Because the names of program objects described by DWARF are the names as they appear in the source program, implementations of language translators that use some form of mangled name (as do many implementations of C++) should use the unmangled form of the name in the DWARF DW_AT_name attribute, including the keyword `operator` *(in names such as* `"operator +"`*), if present. See also Section 2.22 regarding the use of DW_AT_linkage_name for mangled names. Sequences of multiple whitespace characters may be compressed.*

2.16 Data Locations and DWARF Procedures

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

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

2.17 Code Addresses and Ranges

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

- A DW_AT_low_pc attribute for a single address,
- A DW_AT_low_pc and DW_AT_high_pc pair of attributes for a single contiguous range of addresses, or
- A DW_AT_ranges attribute for a non-contiguous range of addresses.

In addition, a non-contiguous range of addresses may also be specified for the DW_AT_start_scope attribute.

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

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 relocated address for the entity.

While the DW_AT_entry_pc attribute might also seem appropriate for this purpose, historically the DW_AT_low_pc attribute was used before the DW_AT_entry_pc was introduced (in DWARF Version 3). There is insufficient reason to change this.

2.17.2 Contiguous Address Range

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

The presence of low and high PC attributes for an entity implies that the code generated for the entity is contiguous and exists totally within the boundaries specified by those two attributes. If that is not the case, no low and high PC attributes should be produced.

2.17.3 Non-Contiguous Address Ranges

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

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

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

A range list entry consists of:

1. A beginning address offset. This address offset has the size of an address and is relative to the applicable base address of the compilation unit referencing this range list. It marks the beginning of an address range.

2. An ending address offset. This address offset again has the size of an address and is relative to the applicable base address of the compilation unit referencing this range list. It marks the first address past the end of the address range.The ending address must be greater than or equal to the beginning address.

   A range list entry (but not a base address selection or end of list entry) whose beginning and ending addresses are equal has no effect because the size of the range covered by such an entry is zero.

The applicable base address of a range list entry is determined by the closest preceding base address selection entry (see below) in the same range list. If there is no such selection entry, then the applicable base address defaults to the base address of the compilation unit (see Section 3.1.1).

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

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

A base address selection entry consists of:

1. The value of the largest representable address offset (for example, 0xffffffff when the size of an address is 32 bits).

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

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

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

A base address selection entry and an end of list entry for a range list are identical to a base address selection entry and end of list entry, respectively, for a location list (see Section 2.6.2) in interpretation and representation.

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, ordinary blocks, try/catch blocks, and the like, may have a DW_AT_entry_pc attribute to indicate the first executable instruction within that range of addresses. The value of the DW_AT_entry_pc attribute is a relocated address. If no DW_AT_entry_pc attribute is present, then the entry address is assumed to be the same as the value of the DW_AT_low_pc attribute, if present; otherwise, the entry address is unknown.

2.19 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 entity which specifies the value of the attribute.
- For an exprloc, the value is interpreted as a DWARF expression; evaluation of the expression yields the value of the attribute.

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

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

2.20 Entity Descriptions

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

Generally, any 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 only display these descriptions as part of the description of other entities. It should not accept them in expressions, nor allow them to be assigned, or the like.

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.

Similarly, 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) 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 describing the object file linkage name associated with the corresponding entity.

Debugging information entries to which DW_AT_linkage_name may apply include: DW_TAG_common_block, DW_TAG_constant, DW_TAG_entry_point, DW_TAG_subprogram and DW_TAG_variable.

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 3.1.3), these entries may be thought of as bounded by ranges of text addresses within the program.

3.1 Unit Entries

An object file may contain one or more compilation units, of which there are three kinds: normal compilation units, partial compilation units and type units. A partial compilation unit is related to one or more other compilation units that import it. A type unit represents a single complete type in a separate unit. Either a normal compilation unit or a partial compilation unit may be logically incorporated into another compilation unit using an imported unit entry.

3.1.1 Normal and Partial Compilation Unit Entries

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

In a simple normal compilation, a single compilation unit with the tag DW_TAG_compile_unit represents a complete object file and the tag DW_TAG_partial_unit is not used. In a compilation employing the DWARF space compression and duplicate elimination techniques from Appendix E.1, multiple compilation units using the tags DW_TAG_compile_unit and/or DW_TAG_partial_unit are used to represent portions of an object file.

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

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

Compilation unit entries may have the following attributes:

1. Either a DW_AT_low_pc and DW_AT_high_pc pair of attributes or a DW_AT_ranges attribute whose values encode the contiguous or non-contiguous address ranges, respectively, of the machine instructions generated for the compilation unit (see Section 2.17).

   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) and range lists (see Section 2.17.3).

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

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

Language Name	Meaning
DW_LANG_Ada83 †	ISO Ada:1983
DW_LANG_Ada95 †	ISO Ada:1995
DW_LANG_C	Non-standardized C, such as K&R
DW_LANG_C89	ISO C:1989
DW_LANG_C99	ISO C:1999
DW_LANG_C_plus_plus	ISO C++:1998
DW_LANG_Cobol74	ISO Cobol:1974
DW_LANG_Cobol85	ISO Cobol:1985
DW_LANG_D †	D
DW_LANG_Fortran77	ISO FORTRAN 77
DW_LANG_Fortran90	ISO Fortran 90

Language Name	Meaning
DW_LANG_Fortran95	ISO Fortran 95
DW_LANG_Java	Java
DW_LANG_Modula2	ISO Modula-2:1996
DW_LANG_ObjC	Objective C
DW_LANG_ObjC_plus_plus	Objective C++
DW_LANG_Pascal83	ISO Pascal:1983
DW_LANG_PLI †	ANSI PL/I:1976
DW_LANG_Python †	Python
DW_LANG_UPC	Unified Parallel C

† Support for these languages is limited.

Figure 8. Language names

4. A DW_AT_stmt_list attribute whose value is a section offset to the line number information for this compilation unit.

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

5. A DW_AT_macro_info attribute whose value is a section offset to the macro information for this compilation unit.

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

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

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

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

DW_ID_case_sensitive
DW_ID_up_case
DW_ID_down_case
DW_ID_case_insensitive

Figure 9. Identifier case codes

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

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

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

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

9. A DW_AT_base_types attribute whose value is a reference.

   This attribute points to a debugging information entry representing another compilation unit. It may be used to specify the compilation unit containing the base type entries used by entries in the current compilation unit (see Section 5.1).

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

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

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

3.1.2 Imported Unit Entries

The place where a normal or partial 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.

3.1.3 Separate 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 a 64-bit 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.

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 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 Figure 8.

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.

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 as it appears in the source program.

The module entry may have either a DW_AT_low_pc and DW_AT_high_pc pair of attributes or a DW_AT_ranges attribute whose values encode the contiguous or non-contiguous address ranges, respectively, of the machine instructions generated for the module initialization code (see Section 2.17). 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).

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.

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

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

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

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

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

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

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

3.2.3 Imported (or Renamed) Declaration Entries

Some languages support the concept of importing into or making accessible in a given unit declarations made in a different module 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, as it appears in the source program, by which the imported entity is to be known in the context of the imported declaration entry (which may be different than the name of the entity being imported). If no name is present, then the name by which the entity is to be known is the same as the name of the entity being imported.

An imported declaration entry with a name attribute may be used as a general means to rename or provide an alias for an entity, regardless of the context in which the importing declaration or the imported entity 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 as it appears in the source program and an import attribute whose value is a reference to the applicable original namespace or namespace extension entry.

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

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

3.2.4 Imported Module Entries

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

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

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 A in module A because there is no explicit representation for X in module B.

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

3.3 Subroutine and Entry Point Entries

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

DW_TAG_subprogram	A 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 as it appears in the source program. It may also have a DW_AT_linkage_name attribute as described in Section 2.22.

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 5.5.7.

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.

Fortran has a PROGRAM statement which is used to specify and provide a user-supplied name for the main subroutine of a 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.

A subroutine entry may contain a DW_AT_calling_convention attribute, whose value is an integer constant. The set of calling convention codes is given in Figure 10.

DW_CC_normal
DW_CC_program
DW_CC_nocall

Figure 10. Calling convention codes

If this attribute is not present, or its value is the constant DW_CC_normal, then the subroutine may be safely called by obeying the "standard" calling conventions of the target architecture. If the value of the calling convention attribute is the constant DW_CC_nocall, the subroutine does not obey standard calling conventions, and it may not be safe for the debugger to call this subroutine.

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

The DW_CC_program value is intended to support Fortran main programs 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.

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 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.*

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.

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.

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).

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).

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

While the DW_AT_entry_pc attribute might also seem appropriate for this purpose, historically the DW_AT_low_pc attribute was used before the DW_AT_entry_pc was introduced (in DWARF Version 3). There is insufficient reason to change this.

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

A subroutine entry representing a subroutine declaration that is not also a definition does not have code address or range attributes.

3.3.4 Declarations Owned by Subroutines and Entry Points

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

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

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

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

3.3.5 Low-Level Information

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

A subroutine or entry point entry may also have a DW_AT_frame_base attribute, whose value is a location description that computes the "frame base" for the subroutine or entry point. 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 procedure is typically an address fixed relative to the first unit of storage allocated for the procedure's stack frame. The DW_AT_frame_base attribute can be used in several ways:

1. *In procedures that need location lists to locate local variables, the DW_AT_frame_base can hold the needed location list, while all variables' location descriptions can be simpler 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 should obey the following constraints:

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

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

3.3.6 Types Thrown by Exceptions

In C++ a subroutine may declare a set of types which it may validly throw.

If a subroutine explicitly declares that it may throw an exception for one or more types, each such type is represented by a debugging information entry with the tag DW_TAG_thrown_type. Each such entry is a child of the entry representing the subroutine that may throw this type. Each thrown type entry contains a DW_AT_type attribute, whose value is a reference to an entry describing the type of the exception that may be thrown.

3.3.7 Function Template Instantiations

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

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

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

2. The subprogram entry and each of its child entries reference a template type parameter entry in any circumstance where the template definition referenced a formal parameterized type.

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

4. 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 Figure 11.

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

Figure 11. Inline codes

In C++, a function or a constructor declared with `constexpr` *is implicitly declared inline. The abstract inline instance (see below) 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.*

3.3.8.1 Abstract Instances

Any debugging information entry that is owned (either directly or indirectly) by a debugging information entry that contains the DW_AT_inline attribute is referred to as an "abstract instance entry." Any subroutine entry that contains a DW_AT_inline attribute whose value is other than DW_INL_not_inlined is known as an "abstract instance root." Any set of abstract instance entries that are all children (either directly or indirectly) of some abstract instance root, together with the root itself, is known as an "abstract instance tree." However, in the case where an abstract instance tree is nested within another abstract instance tree, the entries in the nested abstract instance tree are not considered to be entries in the outer abstract instance tree.

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. This simplifies the following descriptions.

A debugging information entry that is a member of an abstract instance tree should not contain any attributes which describe aspects of the subroutine which vary between distinct 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 for a contrary example.

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

3.3.8.2 Concrete Inlined Instances

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

Each inlined subroutine entry may have either a DW_AT_low_pc and DW_AT_high_pc pair of attributes or a DW_AT_ranges attribute whose values encode the contiguous or non-contiguous address ranges, respectively, of the machine instructions generated for the inlined subroutine (see Section 2.17). 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).

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).

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 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 should refer to the file, line and column of the original declaration of the subroutine, not to the point at which it was inlined. As a consequence, they may usually be omitted from any entry that has an abstract origin attribute.

For each pair of entries that are associated via a DW_AT_abstract_origin attribute, both members of the pair 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 should refer to the abstract instance entry.

2. Entries in the concrete instance tree which are associated with entries in the abstract instance tree such that neither has a DW_AT_name attribute, and neither is referenced by any other debugging information entry, may be omitted. This may happen for debugging information entries in the abstract instance trees that became unnecessary in the concrete instance tree because of additional information available there. For example, an anonymous variable might have been created and described in the abstract instance tree, but because of the actual parameters for a particular 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, should not contain DW_AT_abstract_origin attributes, and must contain all their own attributes directly. This allows an abstract instance tree to omit debugging information entries for anonymous entities that are unlikely to be needed in most 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, 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 or 3.3.8.3, respectively, and without regard to the fact that it is within an outer concrete instance tree.

See Appendix D.7 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, which are listed in order of preference:

- If the value is of class reference, then the value specifies the debugging information entry of the target subprogram.
- If the value is of class address, then the value is the relocated address of the target subprogram.
- If the value is of class string, then the value is the (possibly mangled) name of the target subprogram.
- If the value is of class flag, then the value *true* indicates that the containing subroutine is a trampoline but that the target subroutine is not known.

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

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

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

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

3.4 Lexical Block Entries

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

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

The lexical block entry may have either a DW_AT_low_pc and DW_AT_high_pc pair of attributes or a DW_AT_ranges attribute whose values encode the contiguous or non-contiguous address ranges, respectively, of the machine instructions generated for the lexical block (see Section 2.17).

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

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

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

3.5 Label Entries

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

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

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

3.6 With Statement Entries

Both Pascal and Modula-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).

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

3.7 Try and Catch Block Entries

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

A try block is represented by a debugging information entry with the tag DW_TAG_try_block. A catch block is represented by a debugging information entry with the tag DW_TAG_catch_block.

Both try and catch block entries may have either a DW_AT_low_pc and DW_AT_high_pc pair of attributes or a DW_AT_ranges attribute whose values encode the contiguous or non-contiguous address ranges, respectively, of the machine instructions generated for the block (see Section 2.17).

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.

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 as it appears in the source program.

   If a variable entry describes an anonymous union, the name attribute is omitted or 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.

   In a variable entry representing the definition of a variable (that is, with no DW_AT_declaration attribute) if no location attribute is present, or if the location attribute is present but has an empty location description (as described in Section 2.6), the variable is assumed to exist in the source code but not in the executable program (but see number 10, below).

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

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

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

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

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

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

7. 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 is a reference to the debugging information entry for a variable or subroutine, or the value may be a constant. If the attribute form is of class reference, the default value of the parameter is the value of the referenced variable (which may be constant) or the value returned by the referenced subroutine; a reference value of 0 means that no default value has been specified. If the value is of class constant, that constant is interpreted as a default 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_start_scope attribute if the scope of an object is smaller than (that is, is a subset of the addresses of) the scope most closely enclosing the object. There are two cases:

    a) If the scope of the object entry includes all of the containing scope except for a contiguous sequence of bytes at the beginning of that containing scope, then the scope of the object is specified using a value of class constant. If the containing scope is contiguous, the value of this attribute is the offset in bytes of the beginning of the scope for the object from the low pc value of the debugging information entry that defines its scope. If the containing scope is non-contiguous (see Section 2.17.3), the value of this attribute is the offset in bytes of the beginning of the scope for the object from the beginning of the first range list entry that is not a base selection entry or an end of list entry.

    b) Otherwise, the scope of the object is specified using a value of class rangelistptr. This value indicates the beginning of a range list (see Section 2.17.3).

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

```
float x = 99.99;

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

    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.

12. 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 Figure 12.

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

Figure 12. Endianity attribute values

    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.

13. A DW_AT_const_expr attribute, which is a flag, if a variable entry represents a C++ object declared with the `constexpr` specifier. This attributes 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.*

14. A DW_AT_linkage_name attribute for a variable or constant entry as described in Section 2.22.

4.2 Common Block Entries

A Fortran common block may be described by a debugging information entry with the tag DW_TAG_common_block. The common block entry has a DW_AT_name attribute whose value is a null-terminated string containing the common block name as it appears in the source program. It may also have a DW_AT_linkage_name attribute as described in Section 2.22. It also has a DW_AT_location attribute whose value describes the location of the beginning of the common block. The common block entry owns debugging information entries describing the variables contained within the common block.

4.3 Namelist Entries

At least one language, Fortran 90, has the concept of a namelist. A namelist is an ordered list of the names of some set of declared objects. The namelist object itself may be used as a replacement for the list of names in various contexts.

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

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

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

5 TYPE ENTRIES

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

If the scope of the declaration of a named type begins after the low pc value for the scope most closely enclosing the declaration, the declaration may have a DW_AT_start_scope attribute as described for objects in Section 4.1.

5.1 Base Type Entries

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

A base type is represented by a debugging information entry with the tag DW_TAG_base_type.

A base type entry has a DW_AT_name attribute whose value is a null-terminated string 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 value of this attribute is an integer constant. The set of values and their meanings for the DW_AT_encoding attribute is given in Figure 13 and following text.

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

A base type entry has either a DW_AT_byte_size attribute or a DW_AT_bit_size attribute whose integer constant value (see Section 2.21) 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). 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. The data bit offset uses the bit numbering and direction conventions that are appropriate to the current language on the

target system to locate the beginning of the storage and value. If this attribute is omitted a default data bit offset of zero is assumed.

Attribute DW_AT_data_bit_offset is new in DWARF Version 4 and is also used for bit field members (see Section 5.5.6). It replaces the attribute DW_AT_bit_offset when used for base types as defined in DWARF V3 and earlier. The earlier attribute is defined in a manner suitable for bit field members on big-endian architectures but which is wasteful for use on little-endian architectures.

The attribute DW_AT_bit_offset is deprecated in DWARF Version 4 for use in base types, but implementations may continue to support its use for compatibility.

The DWARF Version 3 definition of these attributes is as follows.

> *A base type entry has a DW_AT_byte_size attribute, whose value (see Section 2.19) is the size in bytes of the storage unit used to represent an object of the given type.*
>
> *If the value of an object of the given type does not fully occupy the storage unit described by the byte size attribute, the base type entry may have a DW_AT_bit_size attribute and a DW_AT_bit_offset attribute, both of whose values (see Section 2.19) are integers. The bit size attribute describes the actual size in bits used to represent a value of the given type. The bit offset attribute describes the offset in bits of the high order bit of a value of the given type from the high order bit of the storage unit used to contain that value.*

In comparing DWARF Versions 3 and 4, note that DWARF V4 defines the following combinations of attributes:

- *DW_AT_byte_size*
- *DW_AT_bit_size*
- *DW_AT_byte_size, DW_AT_bit_size and optionally DW_AT_data_bit_offset*

DWARF V3 defines the following combinations:

- *DW_AT_byte_size*
- *DW_AT_byte_size, DW_AT_bit_size and DW_AT_bit_offset*

Name	Meaning
DW_ATE_address	linear machine address (for segmented addresses see Section 2.12)
DW_ATE_boolean	true or false
DW_ATE_complex_float	complex binary floating-point number
DW_ATE_float	binary floating-point number
DW_ATE_imaginary_float	imaginary binary floating-point number
DW_ATE_signed	signed binary integer
DW_ATE_signed_char	signed character
DW_ATE_unsigned	unsigned binary integer
DW_ATE_unsigned_char	unsigned character
DW_ATE_packed_decimal	packed decimal
DW_ATE_numeric_string	numeric string
DW_ATE_edited	edited string
DW_ATE_signed_fixed	signed fixed-point scaled integer
DW_ATE_unsigned_fixed	unsigned fixed-point scaled integer
DW_ATE_decimal_float	decimal floating-point number
DW_ATE_UTF	Unicode character

Figure 13. Encoding attribute values

The DW_ATE_decimal_float encoding is intended for floating-point representations that have a power-of-ten exponent, such as that specified in IEEE 754R.

The DW_ATE_UTF encoding is intended for Unicode string encodings (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.

The DW_ATE_packed_decimal and DW_ATE_numeric_string base types 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 Figure 14). 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.

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_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_ATE_edited base type is used to represent an edited numeric or alphanumeric data type. It is used in combination with an 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.

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 entries 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 types.

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.

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.*

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	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

Figure 14. Decimal sign attribute values

5.2 Unspecified Type Entries

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

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

The interpretation of this debugging information entry is intentionally left flexible to allow it to be interpreted appropriately in different languages. For example, in C and C++ the language implementation can provide an unspecified type entry with the name “void” which can be referenced by the type attribute of pointer types and typedef declarations for 'void' (see Sections 0 and 5.3, 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). Type Modifier Entries

A base or user-defined type may be modified in different ways in different languages. A type modifier is represented in DWARF by a debugging information entry with one of the tags given in Figure 15.

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

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

A modified type entry describing a pointer or reference type (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 ought to be dereferenced.

A modified type entry describing a shared qualified type (using DW_TAG_shared_type) may have a DW_AT_count attribute whose value is a constant expressing the blocksize of the type. If no count attribute is present, then the "infinite" blocksize is assumed.

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

Tag	Meaning
DW_TAG_const_type	C or C++ const qualified type
DW_TAG_packed_type	Pascal or Ada packed type
DW_TAG_pointer_type	Pointer to an object of the type being modified.
DW_TAG_reference_type	C++ (lvalue) reference to an object of the type being modified
DW_TAG_restrict_type	C restrict qualified type
DW_TAG_rvalue_reference_type	C++ rvalue reference to an object of the type being modified
DW_TAG_shared_type	UPC shared qualified type
DW_TAG_volatile_type	C or C++ volatile qualified type

Figure 15. Type modifier tags

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

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


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

5.3 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 as it appears in the source program.

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.

5.4 Array Type Entries

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

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

The array type entry describing a multidimensional array may have a DW_AT_ordering attribute whose integer constant value is interpreted to mean either row-major or column-major ordering of array elements. The set of values and their meanings for the ordering attribute are listed in Figure 16. If no ordering attribute is present, the default ordering for the source language (which is indicated by the DW_AT_language attribute of the enclosing compilation unit entry) is assumed.

DW_ORD_col_major
DW_ORD_row_major

Figure 16. Array ordering

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

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

If the amount of storage allocated to hold each element of an object of the given array type is different from the amount of storage that is normally allocated to hold an individual object of the indicated element type, then the array type entry has either a DW_AT_byte_stride or a DW_AT_bit_stride attribute, whose value (see Section 2.19) 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), 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 (i.e., leftmost dimension first, next to leftmost second, and so on).

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

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

5.5 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.

5.5.1 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 as it appears in the source program.

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

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), 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 3.1.3), an incomplete declaration of that type in the compilation unit may provide the unique 64-bit signature of the type using a DW_AT_signature attribute.

If a structure, union or class entry represents the definition of a structure, class or union member corresponding to a prior incomplete structure, class or union, 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, class or union 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*).*

5.5.2 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 as it appears in the source program.

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

5.5.3 Derived or Extended Structs, 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 derived class. If that value is a constant, it is the offset in bytes from the beginning of the class to the beginning of the inherited type. Otherwise, the value must be a location description. In this latter case, the beginning address of the derived class is pushed on the expression stack before the location description is evaluated and the result of the evaluation is the location of the inherited type.

The interpretation of the value of this attribute for inherited types is the same as the interpretation for data members (see Section 5.5.6).

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

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.

5.5.4 Access Declarations

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

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

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

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

5.5.5 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.

5.5.6 Data Member Entries

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

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

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

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.

Bit offsets in DWARF use the bit numbering and direction conventions that are appropriate to the current language on the target system.

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); 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) 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) is the amount of storage needed to hold the value of the data member.

C and C++ bit fields 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.*

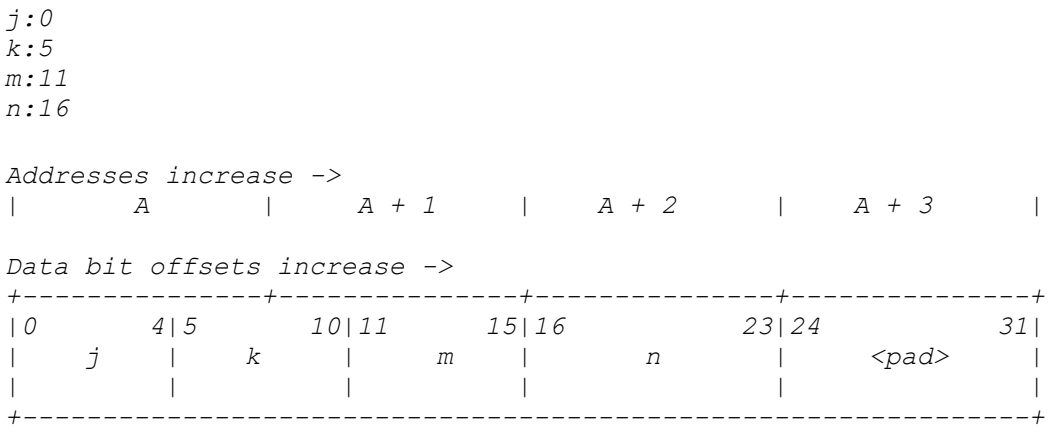
In either case, the bit so identified is defined as the beginning of the object.

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

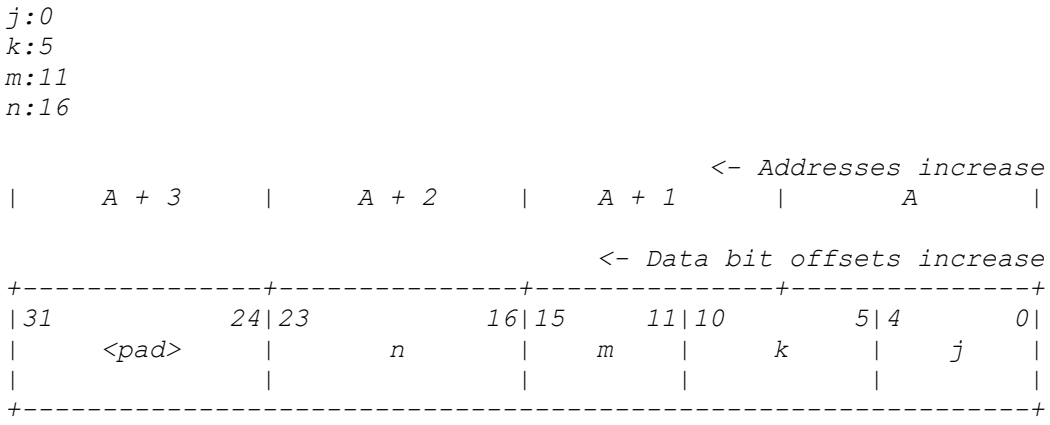
```
struct S {
    int j:5;
    int k:6;
    int m:5;
    int n:8;
};
```

The following diagrams 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.

Big-Endian Data Bit Offsets:



Little-Endian Data Bit Offsets:



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.

For a more extensive example showing nested and packed records and arrays, see Appendix D.2.3.

Attribute DW_AT_data_bit_offset is new in DWARF Version 4 and is also used for base types (see Section 5.1). It replaces the attributes DW_AT_bit_offset and DW_AT_byte_size when used to identify the beginning of bit field data members as defined in DWARF V3 and earlier. The earlier attributes are defined in a manner suitable for bit field members on big-endian architectures but which is either awkward or incomplete for use on little-endian architectures. (DW_AT_byte_size also has other uses that are not affected by this change.)

The DW_AT_byte_size, DW_AT_bit_size and DW_AT_bit_offset attribute combination is deprecated for data members in DWARF Version 4, but implementations may continue to support this use for compatibility.

The DWARF Version 3 definitions of these attributes are as follows.

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

- *A DW_AT_byte_size attribute whose value (see Section 2.19) is the number of bytes that contain an instance of the bit field and any padding bits.*

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

- *A DW_AT_bit_offset attribute whose value (see Section 2.19) is the number of bits to the left of the leftmost (most significant) bit of the bit field value.*

- *A DW_AT_bit_size attribute whose value (see Section 2.19) is the number of bits occupied by the bit field value.*

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

Diagrams similar to the above that show the use of the DW_AT_byte_size, DW_AT_bit_size and DW_AT_bit_offset attribute combination may be found in the DWARF Version 3 Standard.

In comparing DWARF Versions 3 and 4, note that DWARF V4 defines the following combinations of attributes:

- *either DW_AT_data_member_location or DW_AT_data_bit_offset (to specify the beginning of the data member)*

optionally together with

- *either DW_AT_byte_size or DW_AT_bit_size (to specify the size of the data member)*

DWARF V3 defines the following combinations

- *DW_AT_data_member_location (to specify the beginning of the data member, except this specification is only partial in the case of a bit field)*

optionally together with

- *DW_AT_byte_size, DW_AT_bit_size and DW_AT_bit_offset (to further specify the beginning of a bit field data member as well as specify the size of the data member)*

5.5.7 Member Function Entries

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

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

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

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.

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.

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

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

5.5.8 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 both parameterized types and parameterized constant values. DWARF does not represent the generic template definition, but does represent each instantiation.

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

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

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

3. The class type entry and each of its child entries references a template type parameter entry in any circumstance where the source template definition references a formal parameterized type. Similarly, the class type entry and each of its child entries references a template value parameter entry in any circumstance where the source template definition references a formal parameterized value.

4. 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 should be empty or omitted.

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

5.5.9 Variant Entries

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

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

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

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

Alternatively, the variant entry may contain a DW_AT_discr_list attribute, whose value represents a list of discriminant values. This list is represented by any of the block forms and may contain a mixture of case labels and label ranges. Each item on the list is prefixed with a discriminant value descriptor that determines whether the list item represents a single label or a label range. A single case label is represented as an LEB128 number as defined above for the DW_AT_discr_value attribute. A label range is represented by two LEB128 numbers, the low value of the range followed by the high value. Both values follow the rules for signedness just described. The discriminant value descriptor is an integer constant that may have one of the values given in Figure 17.

Figure 17. Discriminant descriptor values

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

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

5.6 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 as it appears in the source program.

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 should describe 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.

5.7 Enumeration Type Entries

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

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

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

The enumeration type entry may have a DW_AT_type attribute which refers to the underlying data type used to implement the enumeration.

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 as it appears in the source program. Each enumerator entry also has a DW_AT_const_value attribute, whose value is the actual numeric value of the enumerator as represented on the target system.

If the enumeration type occurs as the description of a dimension of an array type, and the stride for that dimension is different than what would otherwise be determined, then the enumeration type entry has 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. 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.

5.8 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 as it appears in the source program.

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

In 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 has a tag whose value has one of two possible interpretations:

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

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

5.9 String Type Entries

A "string" is a sequence of characters that have specific semantics and operations that separate them from arrays of characters. Fortran is one of the languages that has a string type. Note that "string" in this context refers to a target machine concept, not the class string as used in this document (except for the name attribute).

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

The string type entry may have a DW_AT_string_length attribute whose value is a location description yielding the location where the length of the string is stored in the program. The string type entry may also have a DW_AT_byte_size attribute or DW_AT_bit_size attribute, whose value (see Section 2.21) is the size of the data to be retrieved from the location referenced by the 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.

If no string length attribute is present, the string type entry may have a DW_AT_byte_size attribute or DW_AT_bit_size attribute, whose value (see Section 2.21) is the amount of storage needed to hold a value of the string type.

5.10 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 as it appears in the source program.

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

If the amount of storage allocated to hold each element of an object of the given set type is different from the amount of storage that is normally allocated to hold an individual object of the indicated element type, then the set type entry has either a DW_AT_byte_size attribute, or DW_AT_bit_size attribute whose value (see Section 2.21) is the amount of storage needed to hold a value of the set type.

5.11 Subrange Type Entries

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

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

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

If the amount of storage allocated to hold each element of an object of the given subrange type is different from the amount of storage that is normally allocated to hold an individual object of the indicated element type, then the subrange type entry has a DW_AT_byte_size attribute or DW_AT_bit_size attribute, whose value (see Section 2.19) 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.

If the lower bound value is missing, the value is assumed to be a language-dependent default constant. The default lower bound is 0 for C, C++, D, Java, Objective C, Objective C++, Python, and UPC. The default lower bound is 1 for Ada, COBOL, Fortran, Modula-2, Pascal and PL/I.

No other default lower bound values are currently defined.

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

If the subrange entry has no type attribute describing the basis type, the basis type is assumed to be the same as the object described by the lower bound attribute (if it references an object). If there is no lower bound attribute, or that attribute does not reference an object, the basis type is the type of the upper bound or count attribute (if either of them references an object). If there is no upper bound or count attribute, or neither references an object, the type is assumed to be the same type, in the source language of the compilation unit containing the subrange entry, as a signed integer with the same size as an address on the target machine.

If the subrange type occurs as the description of a dimension of an array type, and the stride for that dimension is different than what would otherwise be determined, then the subrange type entry has 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..

Note that the stride can be negative.

5.12 Pointer to Member Type Entries

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

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

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

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

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

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`*.*

5.13 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 as it appears in the source program.

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

The file type entry also has a DW_AT_byte_size or DW_AT_bit_size attribute, whose value (see Section 2.19) is the amount of storage need to hold a value of the file type.

5.14 Dynamic Type Properties

5.14.1 Data Location

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

The DW_AT_data_location attribute may be used with any type that provides one or more levels of hidden indirection and/or run-time parameters in its representation. Its value is a location 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.

5.14.2 Allocation and Association Status

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

The DW_AT_allocated attribute may optionally be used with any type for which objects of the type can be explicitly allocated and deallocated. The presence of the attribute indicates that objects of the type are allocatable and deallocatable. The 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.

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

The value of these attributes is determined as described in Section 2.19.

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.

5.15 Template Alias Entries

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 as it appears in the source program. The template alias entry also contains a DW_AT_type attribute whose value is a reference to the type named by the template alias. The template alias entry has the following child entries:

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

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` or `.debug_types` 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 and 7.4.
- Sbyte, ubyte, uhalf, and uword, which are defined in Section 7.26.

6.1 Accelerated Access

A debugger frequently needs to find the debugging information for a program entity defined outside of the compilation unit where the debugged program is currently stopped. Sometimes 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 (data objects, functions and types) by name or by address faster, a producer of DWARF information may provide three different types of tables containing information about the debugging information entries owned by a particular compilation unit entry in a more condensed format.

6.1.1 Lookup by Name

For lookup by name, two tables are maintained in separate object file sections named `.debug_pubnames` for objects and functions, and `.debug_pubtypes` for types. Each table consists of sets of variable length entries. Each set describes the names of global objects and functions, or global types, respectively, whose definitions are represented by debugging information entries owned by a single compilation unit.

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

Each set begins with a header containing four values:

1. `unit_length` (initial length)

   The total length of the all of the entries for that set, not including the length field itself (see Section 7.2.2).

2. `version` (uhalf)

   A version number (see Section 7.19). This number is specific to the name lookup table and is independent of the DWARF version number.

3. `debug_info_offset` (section offset)

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

4. `debug_info_length` (section length)

   The size in bytes of the contents of the `.debug_info` section generated to represent that compilation unit.

This header is followed by a variable number of offset/name pairs. Each pair consists of the section offset from the beginning of the compilation unit corresponding to the current set to the debugging information entry for the given object, followed by a null-terminated character string representing the name of the object as given by the DW_AT_name attribute of the referenced debugging information entry. Each set of names is terminated by an offset field containing zero (and no following string).

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

6.1.2 Lookup by Address

For lookup by address, a table is maintained in a separate object file section called `.debug_aranges`. The table consists of sets of variable length entries, each set describing the portion of the program's address space that is covered by a single compilation unit.

Each set begins with a header containing five values:

1. `unit_length` (initial length)

   The total length of all of the entries for that set, not including the length field itself (see Section 7.2.2).

2. `version` (uhalf)

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

Line number information generated for a compilation unit is represented in the `.debug_line` section of an object file and is referenced by a corresponding compilation unit debugging information entry (see Section 3.1.1) 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 information is identical with that of its predecessors. Any deleted row would never be the beginning of a source statement. Second, we design a byte-coded language for a state machine and store a stream of bytes in the object file instead of the matrix. This language can be much more compact than the matrix. When a consumer of the line number information executes, it must "run" the state machine to generate the matrix for each compilation unit it is interested in. The concept of an encoded matrix also leaves room for expansion. In the future, columns can be added to the matrix to encode other things that are related to individual instruction addresses.

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

6.2.1 Definitions

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

state machine	The hypothetical machine used by a consumer of the line number information to expand the byte-coded instruction stream into a matrix of line number information.
line number program	A series of byte-coded line number information instructions representing one compilation unit.
basic block	A sequence of instructions where only the first instruction may be a branch target and only the last instruction may transfer control. A procedure 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 the following registers:

`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. The `address` and `op_index` registers, taken together, form an *operation pointer* that can reference any individual operation with the instruction stream.
`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 an entry breakpoint of a function.
`epilogue_begin`	A boolean indicating that the current address is one (of possibly many) where execution should be suspended for an exit breakpoint of a function.

`isa` An unsigned integer whose value encodes the applicable instruction set architecture for the current instruction.

The encoding of instruction sets should be shared by all users of a given architecture. It is recommended that this encoding be defined by the ABI authoring committee for each architecture.

`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 should be zero.

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

`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`

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:

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

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

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

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

6.2.4 The Line Number Program Header

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

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

1. `unit_length` (initial length)

   The size in bytes of the line number information for this compilation unit, not including the `unit_length` field itself (see Section 7.2.2).

2. `version` (uhalf)

   A version number (see Appendix F). This number is specific to the line number information and is independent of the DWARF version number.

3. `header_length`

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

4. `minimum_instruction_length` (ubyte)

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

5. `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.

6. `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.

7. `line_base` (sbyte)

   This parameter affects the meaning of the special opcodes. See below.

8. `line_range` (ubyte)

   This parameter affects the meaning of the special opcodes. See below.

9. `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 Version 3 and Version 4, 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.

10. `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.

11. `include_directories` (sequence of path names)

    Entries in this sequence describe each path that was searched for included source files in this compilation. (The paths include those directories specified explicitly by the user for the compiler to search and those the compiler searches without explicit direction.) Each path entry is either a full path name or is relative to the current directory of the compilation.

    The last entry is followed by a single null byte.

    The line number program assigns numbers to each of the file entries in order, beginning with 1. The current directory of the compilation is understood to be the zeroth entry and is not explicitly represented.

12. `file_names` (sequence of file entries)

    Entries in this sequence describe source files that contribute to the line number information for this compilation unit or is used in other contexts, such as in a declaration coordinate or a macro file inclusion. Each entry consists of the following values:

    - A null-terminated string containing the full or relative path name of a source file. If the entry contains a file name or relative path name, the file is located relative to either the compilation directory (as specified by the DW_AT_comp_dir attribute given in the compilation unit) or one of the directories listed in the `include_directories` section.
    - An unsigned LEB128 number representing the directory index of a directory in the `include_directories` section.
    - An unsigned LEB128 number representing the (implementation-defined) time of last modification for the file, or 0 if not available.
    - An unsigned LEB128 number representing the length in bytes of the file, or 0 if not available.

    The last entry is followed by a single null byte.

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

    The primary source file is described by an entry whose path name exactly matches that given in the DW_AT_name attribute in the compilation unit, and whose directory is understood to be given by the implicit entry with index 0.

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

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

6.2.5 The Line Number Program

As stated before, the goal of a line number program is to build a matrix representing one compilation unit, which may have produced multiple sequences of target machine instructions. Within a sequence, addresses (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 now ranges from 13 through 255, the lower bound may increase if one adds new standard opcodes. Thus, the `opcode_base` field of the line number program header gives the value of the first special opcode. Second, the best choice of special-opcode meanings depends on the target architecture. For example, for a RISC machine where the compiler-generated code interleaves instructions from different lines to schedule the pipeline, it is important to be able to add a negative value to the `line` register to express the fact that a later instruction may have been emitted for an earlier source line. For a machine where pipeline scheduling never occurs, it is advantageous to trade away the ability to decrease the line register (a standard opcode provides an alternate way to decrease the line number) in return for the ability to add larger positive values to the `address` register. To permit this variety of strategies, the line number program header defines a `line_base` field that specifies the minimum value which a special opcode can add to the `line` register and a `line_range` field that defines the range of values it can add to the `line` register.

A special opcode value is chosen based on the amount that needs to be added to the `line`, `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.

The amount to increment the `line` register is the `line_base` plus the result of the adjusted opcode modulo the `line_range`. That is,

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

As an example, suppose that the `opcode_base` *is 13,* `line_base` *is -3,* `line_range` *is 12, minimum_instruction_length is 1 and maximum_operations_per_instruction is 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], but not all line advances are available for the maximum operation advance (see below).*

The opcode mapping would be:

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									

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

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.

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.

   Existing assemblers cannot emit DW_LNS_advance_pc or special opcodes because they cannot encode LEB128 numbers or judge when the computation of a special opcode overflows and requires the use of DW_LNS_advance_pc. Such assemblers, however, can use DW_LNS_fixed_advance_pc instead, sacrificing compression.

10. **DW_LNS_set_prologue_end**

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

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

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

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

11. DW_LNS_set_epilogue_begin

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

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

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

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

12. DW_LNS_set_isa

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

6.2.5.3 Extended Opcodes

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). 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_define_file**

   The DW_LNE_define_file opcode takes four operands:

   1. A null-terminated string containing the full or relative path name of a source file. If the entry contains a file name or a relative path name, the file is located relative to either the compilation directory (as specified by the DW_AT_comp_dir attribute given in the compilation unit) or one of the directories in the `include_directories` section.
   2. An unsigned LEB128 number representing the directory index of the directory in which the file was found.
   3. An unsigned LEB128 number representing the time of last modification of the file, or 0 if not available.
   4. An unsigned LEB128 number representing the length in bytes of the file, or 0 if not available.

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

   The primary source file is described by an entry whose path name exactly matches that given in the DW_AT_name attribute in the compilation unit, and whose directory index is 0. The files are numbered, starting at 1, in the order in which they appear; the names in the header come before names defined by the DW_LNE_define_file instruction. These numbers are used in the `file` register of the state machine.

4. **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.

Appendix D.5 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, the macro information for a given compilation unit is represented in the `.debug_macinfo` section of an object file. The macro information for each compilation unit is represented as a series of "macinfo" entries. Each macinfo entry consists of a "type code" and up to two additional operands. The series of entries for a given compilation unit ends with an entry containing a type code of 0.

6.3.1 Macinfo Types

The valid macinfo types are as follows:

DW_MACINFO_define	A macro definition.
DW_MACINFO_undef	A macro undefinition.
DW_MACINFO_start_file	The start of a new source file inclusion.
DW_MACINFO_end_file	The end of the current source file inclusion.
DW_MACINFO_vendor_ext	Vendor specific macro information directives.

6.3.1.1 Define and Undefine Entries

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

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

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

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

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

6.3.1.2 Start File Entries

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

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

6.3.1.3 End File Entries

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

6.3.1.4 Vendor Extension Entries

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

A consumer must be able to totally ignore all DW_MACINFO_vendor_ext entries that it does not understand (see Section 7.1).

6.3.2 Base Source Entries

A producer shall generate DW_MACINFO_start_file and DW_MACINFO_end_file entries for the source file submitted to the compiler for compilation. This DW_MACINFO_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 Macinfo Entries for Command Line Options

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

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

6.3.4 General Rules and Restrictions

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

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

6.4 Call Frame Information

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

- *A code location that is within the subroutine. This location is either the place where the program stopped when the debugger got control (e.g. a breakpoint), or is a place where a subroutine made a call or was interrupted by an asynchronous event (e.g. a signal).*
- *An area of memory that is allocated on a stack called a "call frame." The call frame is identified by an address on the stack. We refer to this address as the Canonical Frame Address or CFA. Typically, the CFA is defined to be the value of the stack pointer at the call site in the previous frame (which may be different from its value on entry to the current frame).*
- *A set of registers that are in use by the subroutine at the code location.*

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

To be able to view or modify an activation that is not on the top of the call frame stack, the debugger must "virtually unwind" the stack of activations until it finds the activation of interest. A debugger unwinds a stack in steps. Starting with the current activation it 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 unwinding operation needs to know where registers are saved and how to compute the predecessor's CFA and code location. When considering an architecture-independent way of encoding this information one has to consider a number of special things.

- *Prologue and epilogue code is not always in distinct blocks at the beginning and end of a subroutine. It is common to duplicate the epilogue code at the site of each return from the code. Sometimes a compiler breaks up the register save/unsave operations and moves them into the body of the subroutine to just where they are needed.*

- *Compilers use different ways to manage the call frame. Sometimes they use a frame pointer register, sometimes not.*
- *The algorithm to compute CFA changes as you progress through the prologue and epilogue code. (By definition, the CFA value does not change.)*
- *Some subroutines have no call frame.*
- *Sometimes a register is saved in another register that by convention does not need to be saved.*
- *Some architectures have special instructions that perform some or all of the register management in one instruction, leaving special information on the stack that indicates how registers are saved.*
- *Some architectures treat return address values specially. For example, in one architecture, the call instruction guarantees that the low order two bits will be zero and the return instruction ignores those bits. This leaves two bits of storage that are available to other uses that must be treated specially.*

6.4.1 Structure of Call Frame Information

DWARF supports virtual unwinding by defining an architecture independent basis for recording how procedures save and restore registers 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 objects, 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 labeled by register number. This includes some registers that have special designation on some architectures such as the PC and the stack pointer register. (The actual mapping of registers for a particular architecture is 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.
val_expression(E)	The previous value of this register is the value produced by executing the DWARF expression E.
architectural	The rule is defined externally to this specification by the augmenter.

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

The virtual unwind information is encoded in a self-contained section called `.debug_frame`. Entries in a `.debug_frame` section are aligned on 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). 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)

   A constant that is used to distinguish CIEs from FDEs.

3. `version` (ubyte)

   A version number (see Section 7.23). This number is specific to the call frame information and is independent of the DWARF version number.

4. `augmentation` (UTF-8 string)

   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 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_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 (seeSection 6.4.2.1).

8. `data_alignment_factor` (signed LEB128)

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

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_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 below.

6. `padding` (array of ubyte)

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

6.4.2 Call Frame Instructions

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

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

- DW_OP_call2, DW_OP_call4 and DW_OP_call_ref operators are not meaningful in an operand of these instructions because there is no mapping from call frame information to any corresponding debugging compilation unit information, thus there is no way to interpret the call offset.
- DW_OP_push_object_address is 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_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 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 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 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 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), one searches through the FDE headers looking at the `initial_location` *and* `address_range` *values to see if L1 is contained in the FDE. If so, then:*

1. *Initialize a register set by reading the* `initial_instructions` *field of the associated CIE.*
2. *Read and process the FDE's instruction sequence until a DW_CFA_advance_loc, DW_CFA_set_loc, or the end of the instruction stream is encountered.*

3. *If a DW_CFA_advance_loc or DW_CFA_set_loc instruction is encountered, then compute a new location value (L2). If L1 >= L2 then process the instruction and go back to step 2.*

4. *The end of the instruction stream can be thought of as a DW_CFA_set_loc (initial_location + address_range) instruction. 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.

6.4.4 Call Frame Calling Address

When 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 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 (e.g. 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.

7 DATA REPRESENTATION

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

7.1 Vendor Extensibility

To reserve a portion of the DWARF name space and ranges of enumeration values for use for vendor specific extensions, special labels are reserved for tag names, attribute names, base type encodings, location operations, language names, calling conventions and call frame instructions.

The labels denoting the beginning and end of the reserved value range for vendor specific extensions consist of the appropriate prefix (DW_TAG, DW_AT, DW_END, DW_ATE, DW_OP, DW_LANG, DW_LNE, DW_CC or DW_CFA respectively) followed by `_lo_user` or `_hi_user`. *For example, for entry tags, the special labels are DW_TAG_lo_user and DW_TAG_hi_user.* Values in the range between *prefix*`_lo_user` and *prefix*`_hi_user` inclusive, are reserved for vendor specific extensions. Vendors may use values in this range without conflicting with current or future system-defined values. All other values are reserved for use by the system.

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

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

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

1. New attributes should be added in such a way that a debugger may recognize the format of a new attribute value without knowing the content of that attribute value.
2. The semantics of any new attributes should not alter the semantics of previously existing attributes.
3. The semantics of any new tags should not conflict with the semantics of previously existing tags.
4. Do not add any new forms of attribute value.

7.2 Reserved Values

7.2.1 Error Values

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

7.2.2 Initial Length Values

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

In an initial length field, the values `0xfffffff0` through `0xffffffff` are reserved by DWARF to indicate some form of extension relative to DWARF Version 2; such values must not be interpreted as a length field. The use of one such value, `0xffffffff`, is defined below (see Section 7.4); the use of the other values is reserved for possible future extensions.

7.3 Executable Objects and Shared Objects

The relocated addresses in the debugging information for an executable object are virtual addresses and the relocated addresses in the debugging information for a shared object are offsets relative to the start of the lowest region of memory loaded from that shared object.

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

7.4 32-Bit and 64-Bit DWARF Formats

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

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

1. In the 32-bit DWARF format, an initial length field (see Section 7.2.2) is an unsigned 32-bit integer (which must be less than `0xfffffff0`); in the 64-bit DWARF format, an initial length field is 96 bits in size, and has two parts:

   - The first 32-bits have the value `0xffffffff`.
   - The following 64-bits contain the actual length represented as an unsigned 64-bit integer.

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

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

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

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

Form	Role
DW_FORM_ref_addr	offset in `.debug_info`
DW_FORM_sec_offset	offset in a section other than `.debug_info` or `.debug_str`
DW_FORM_strp	offset in `.debug_str`
DW_OP_call_ref	offset in `.debug_info`

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

The 32-bit and 64-bit DWARF format conventions must 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 objects will suffice to assure that the DWARF sections within each individual linked object are less than 4 GBytes in size. However, for those cases where needed, the 64-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) 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 contribution may be made to the `.debug_types` section of the object file. Each such contribution consists of a type unit header (see Section 7.5.1.2) 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` or `.debug_types` 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

7.5.1.1 Compilation Unit Header

The header for the series of debugging information entries contributed by a single normal or partial compilation 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 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).

2. `version` (uhalf)

   A 2-byte unsigned integer representing the version of the DWARF information for the compilation unit (see Appendix F). The value in this field is 4.

3. `debug_abbrev_offset` (section offset)

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

4. `address_size` (ubyte)

   A 1-byte unsigned integer representing the size in bytes of an address on the target architecture. If the system uses segmented addressing, this value represents the size of the offset portion of an address.

7.5.1.2 Type Unit Header

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_types` 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_types` 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).

2. `version` (uhalf)

   A 2-byte unsigned integer representing the version of the DWARF information for the compilation unit (see Appendix F). The value in this field is 4.

3. `debug_abbrev_offset` (section offset)

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

4. `address_size` (ubyte)

   A 1-byte unsigned integer representing the size in bytes of an address on the target architecture. If the system uses segmented addressing, this value represents the size of the offset portion of an address.

5. `type_signature` (8-byte unsigned integer)

   A 64-bit unique signature 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.

6. `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).

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` or `.debug_types` section. As described above, the abbreviation code 0 is reserved for null debugging information entries. The abbreviation code is followed by another unsigned LEB128 number that encodes the entry's tag. The encodings for the tag names are given in Figure 18 (page 154).

Following the tag encoding is a 1-byte value that determines whether a debugging information entry using this abbreviation has child entries or not. If the value is DW_CHILDREN_yes, the next physically succeeding entry of any debugging information entry using this abbreviation is the first child of that entry. If the 1-byte value following the abbreviation's tag encoding is DW_CHILDREN_no, the next physically succeeding entry of any debugging information entry using this abbreviation is a sibling of that entry. (Either the first child or sibling entries may be null entries). The encodings for the child determination byte are given in Figure 19 (page 154). (As mentioned in Section 2.3, each chain of sibling entries is terminated by a null entry.)

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

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

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

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

7.5.4 Attribute Encodings

The encodings for the attribute names are given in Figure 20 (page 159).

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

Form DW_FORM_sec_offset is a member of more than one class, namely lineptr, loclistptr, macptr or rangelistptr; the list of classes allowed by the applicable attribute in Figure 20 determines the class of the form.

In DWARF V3 the forms DW_FORM_data4 and DW_FORM_data8 were members of either class constant or one of the classes lineptr, loclistptr, macptr or rangelistptr, depending on context. In DWARF V4 DW_FORM_data4 and DW_FORM_data8 are members of class constant in all cases. The new DW_FORM_sec_offset replaces their usage for the other classes.

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

address

Represented as an object of appropriate size to hold an address on the target machine (DW_FORM_addr). The size is encoded in the compilation unit header (see Section 7.5.1.1). This address is relocatable in a relocatable object file and is relocated in an executable file or shared object.

block

Blocks come in four forms:

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

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

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

An unsigned LEB128 length followed by the number of bytes specified by the length (DW_FORM_block).

In all forms, the length is the number of information bytes that follow. The information bytes may contain any mixture of relocated (or relocatable) addresses, references to other debugging information entries or data bytes.

constant

There are six forms of constants. There are fixed length constant data forms for one, two, four and eight byte values (respectively, DW_FORM_data1, DW_FORM_data2, DW_FORM_data4, and DW_FORM_data8). There are also variable length constant data forms encoded using LEB128 numbers (see below). Both signed (DW_FORM_sdata) and unsigned (DW_FORM_udata) variable length constants are available

The data in DW_FORM_data1, DW_FORM_data2, DW_FORM_data4 and DW_FORM_data8 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) or location description (see Section 2.6).

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` 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. 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).

loclistptr

This is an offset into the `.debug_loc` section (DW_FORM_sec_offset). It consists of an offset from the beginning of the `.debug_loc` section to the first byte of the data making up the location list for the compilation unit. It is relocatable in a relocatable object file, and relocated in an executable or shared object. 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).

macptr

This is an offset into the `.debug_macinfo` section (DW_FORM_sec_offset). It consists of an offset from the beginning of the `.debug_macinfo` section to the first byte of the data making up the macro information list for the compilation unit. It is relocatable in a relocatable object file, and relocated in an executable or shared object. 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).

rangelistptr

This is an offset into the `.debug_ranges` section (DW_FORM_sec_offset). It consists of an offset from the beginning of the `.debug_ranges` section to the beginning of the non-contiguous address ranges information for the referencing entity. It is relocatable in a relocatable object file and relocated in an executable or shared object. 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).

Because classes lineptr, loclistptr, macptr and rangelistptr share a common representation, it is not possible for an attribute to allow more than one of these classes

reference

There are three 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. This type of reference (DW_FORM_ref_addr) is an offset from the beginning of the `.debug_info` section of the target executable or shared object; it is relocatable in a relocatable object file and frequently relocated in an executable file or shared object. For references from one shared object or static executable file to another, the relocation and identification of the target object must be performed by the consumer. In the 32-bit DWARF

format, this offset is a 4-byte unsigned value; in the 64-bit DWARF format, it is an 8-byte unsigned value (see Section 7.4).

A debugging information entry that may be referenced by another compilation unit using DW_FORM_ref_addr must have a global symbolic name.

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

The 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 64-bit type signature (see Section 7.27) that was computed for the type.

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 and third type of reference allows for the sharing of information, such as types, across compilation units.

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), or may be represented as an offset into a string table contained in the `.debug_str` section of the object file (DW_FORM_strp). In the 32-bit DWARF format, the representation of a DW_FORM_strp value is a 4-byte unsigned offset; in the 64-bit DWARF format, it is an 8-byte unsigned offset (see Section 7.4).

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

The Unicode Standard 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.

In no case does an attribute use one of the classes lineptr, loclistptr, macptr or rangelistptr to point into either the `.debug_info` or `.debug_str` section.

The form encodings are listed in Figure 21 (page 161).

Figure 18, Tag encodings, begins here.

Tag name	Value
DW_TAG_array_type	0x01
DW_TAG_class_type	0x02
DW_TAG_entry_point	0x03
DW_TAG_enumeration_type	0x04
DW_TAG_formal_parameter	0x05
DW_TAG_imported_declaration	0x08
DW_TAG_label	0x0a
DW_TAG_lexical_block	0x0b
DW_TAG_member	0x0d
DW_TAG_pointer_type	0x0f
DW_TAG_reference_type	0x10
DW_TAG_compile_unit	0x11
DW_TAG_string_type	0x12
DW_TAG_structure_type	0x13
DW_TAG_subroutine_type	0x15
DW_TAG_typedef	0x16

Tag name	Value
DW_TAG_union_type	0x17
DW_TAG_unspecified_parameters	0x18
DW_TAG_variant	0x19
DW_TAG_common_block	0x1a
DW_TAG_common_inclusion	0x1b
DW_TAG_inheritance	0x1c
DW_TAG_inlined_subroutine	0x1d
DW_TAG_module	0x1e
DW_TAG_ptr_to_member_type	0x1f
DW_TAG_set_type	0x20
DW_TAG_subrange_type	0x21
DW_TAG_with_stmt	0x22
DW_TAG_access_declaration	0x23
DW_TAG_base_type	0x24
DW_TAG_catch_block	0x25
DW_TAG_const_type	0x26
DW_TAG_constant	0x27
DW_TAG_enumerator	0x28
DW_TAG_file_type	0x29
DW_TAG_friend	0x2a

Tag name	Value
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
DW_TAG_condition	0x3f

Tag name	Value
DW_TAG_shared_type	0x40
DW_TAG_type_unit ‡	0x41
DW_TAG_rvalue_reference_type ‡	0x42
DW_TAG_template_alias ‡	0x43
DW_TAG_lo_user	0x4080
DW_TAG_hi_user	0xffff

‡ New in DWARF Version 4

Figure 18. Tag encodings

Child determination name	Value
DW_CHILDREN_no	0x00
DW_CHILDREN_yes	0x01

Figure 19. Child determination encodings

Figure 20, Attribute encodings, begins here.

Attribute name	Value	Classes
DW_AT_sibling	0x01	reference
DW_AT_location	0x02	exprloc, loclistptr
DW_AT_name	0x03	string
DW_AT_ordering	0x09	constant
DW_AT_byte_size	0x0b	constant, exprloc, reference
DW_AT_bit_offset	0x0c	constant, exprloc, reference
DW_AT_bit_size	0x0d	constant, exprloc, reference
DW_AT_stmt_list	0x10	lineptr
DW_AT_low_pc	0x11	address
DW_AT_high_pc	0x12	address, constant
DW_AT_language	0x13	constant
DW_AT_discr	0x15	reference
DW_AT_discr_value	0x16	constant
DW_AT_visibility	0x17	constant
DW_AT_import	0x18	reference
DW_AT_string_length	0x19	exprloc, loclistptr
DW_AT_common_reference	0x1a	reference
DW_AT_comp_dir	0x1b	string
DW_AT_const_value	0x1c	block, constant, string

Attribute name	Value	Classes
DW_AT_containing_type	0x1d	reference
DW_AT_default_value	0x1e	reference
DW_AT_inline	0x20	constant
DW_AT_is_optional	0x21	flag
DW_AT_lower_bound	0x22	constant, exprloc, reference
DW_AT_producer	0x25	string
DW_AT_prototyped	0x27	flag
DW_AT_return_addr	0x2a	exprloc, loclistptr
DW_AT_start_scope	0x2c	Constant, rangelistptr
DW_AT_bit_stride	0x2e	constant, exprloc, reference
DW_AT_upper_bound	0x2f	constant, exprloc, reference
DW_AT_abstract_origin	0x31	reference
DW_AT_accessibility	0x32	constant
DW_AT_address_class	0x33	constant
DW_AT_artificial	0x34	flag
DW_AT_base_types	0x35	reference
DW_AT_calling_convention	0x36	constant
DW_AT_count	0x37	constant, exprloc, reference
DW_AT_data_member_location	0x38	constant, exprloc, loclistptr
DW_AT_decl_column	0x39	constant

Attribute name	Value	Classes
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, loclistptr
DW_AT_friend	0x41	reference
DW_AT_identifier_case	0x42	constant
DW_AT_macro_info	0x43	macptr
DW_AT_namelist_item	0x44	reference
DW_AT_priority	0x45	reference
DW_AT_segment	0x46	exprloc, loclistptr
DW_AT_specification	0x47	reference
DW_AT_static_link	0x48	exprloc, loclistptr
DW_AT_type	0x49	reference
DW_AT_use_location	0x4a	exprloc, loclistptr
DW_AT_variable_parameter	0x4b	flag
DW_AT_virtuality	0x4c	constant
DW_AT_vtable_elem_location	0x4d	exprloc, loclistptr

Attribute name	Value	Classes
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
DW_AT_use_UTF8	0x53	flag
DW_AT_extension	0x54	reference
DW_AT_ranges	0x55	rangelistptr
DW_AT_trampoline	0x56	address, flag, reference, string
DW_AT_call_column	0x57	constant
DW_AT_call_file	0x58	constant
DW_AT_call_line	0x59	constant
DW_AT_description	0x5a	string
DW_AT_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
DW_AT_mutable	0x61	flag

Attribute name	Value	Classes
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_lo_user	0x2000	---
DW_AT_hi_user	0x3fff	---

‡ New in DWARF Version 4

Figure 20. Attribute encodings

Form name	Value	Class
DW_FORM_addr	0x01	address
DW_FORM_block2	0x03	block
DW_FORM_block4	0x04	block
DW_FORM_data2	0x05	constant
DW_FORM_data4	0x06	constant
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

Form name	Value	Class
DW_FORM_ref_udata	0x15	reference
DW_FORM_indirect	0x16	(see Section 7.5.3)
DW_FORM_sec_offset ‡	0x17	lineptr, loclistptr, macptr, rangelistptr
DW_FORM_exprloc ‡	0x18	exprloc
DW_FORM_flag_present ‡	0x19	flag
DW_FORM_ref_sig8 ‡	0x20	reference

‡ New in DWARF Version 4

Figure 21. Attribute form encodings

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 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.

Figure 22 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 32-bit integer -2. The three high level bytes of the number are sign extension, thus LEB128 would represent it as a single byte containing the low order 7 bits, with the high order bit cleared to indicate the end of the byte stream. Note that there is nothing within the LEB128 representation that indicates whether an encoded number is signed or unsigned. The decoder must know what type of number to expect. Figure 22 gives some examples of unsigned LEB128 numbers and Figure 23 gives some examples of signed LEB128 numbers.

Appendix C gives algorithms for encoding and decoding these forms.

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

Figure 22. Examples of unsigned LEB128 encodings

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

Figure 23. Examples of signed LEB128 encodings

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 Figure 24.

Figure 24, DWARF operation encodings, begins here.

Operation	Code	No. of Operands	Notes
DW_OP_addr	0x03	1	constant address (size target specific)
DW_OP_deref	0x06	0	

Operation	Code	No. of Operands	Notes
DW_OP_const1u	0x08	1	1-byte constant
DW_OP_const1s	0x09	1	1-byte constant
DW_OP_const2u	0x0a	1	2-byte constant
DW_OP_const2s	0x0b	1	2-byte constant
DW_OP_const4u	0x0c	1	4-byte constant
DW_OP_const4s	0x0d	1	4-byte constant
DW_OP_const8u	0x0e	1	8-byte constant
DW_OP_const8s	0x0f	1	8-byte constant
DW_OP_constu	0x10	1	ULEB128 constant
DW_OP_consts	0x11	1	SLEB128 constant
DW_OP_dup	0x12	0	
DW_OP_drop	0x13	0	
DW_OP_over	0x14	0	
DW_OP_pick	0x15	1	1-byte stack index
DW_OP_swap	0x16	0	
DW_OP_rot	0x17	0	
DW_OP_xderef	0x18	0	
DW_OP_abs	0x19	0	
DW_OP_and	0x1a	0	
DW_OP_div	0x1b	0	

Operation	Code	No. of Operands	Notes
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_skip	0x2f	1	signed 2-byte constant
DW_OP_bra	0x28	1	signed 2-byte constant
DW_OP_eq	0x29	0	
DW_OP_ge	0x2a	0	
DW_OP_gt	0x2b	0	
DW_OP_le	0x2c	0	
DW_OP_lt	0x2d	0	
DW_OP_ne	0x2e	0	

Operation	Code	No. of Operands	Notes
DW_OP_lit0	0x30	0	literals 0..31 = (DW_OP_lit0 + literal)
DW_OP_lit1	0x31	0	
...			
DW_OP_lit31	0x4f	0	
DW_OP_reg0	0x50	0	reg 0..31 = (DW_OP_reg0 + regnum)
DW_OP_reg1	0x51	0	
...			
DW_OP_reg31	0x6f	0	
DW_OP_breg0	0x70	1	SLEB128 offset base register 0..31 = (DW_OP_breg0 + regnum)
DW_OP_breg1	0x71	1	
...			
DW_OP_breg31	0x8f	1	
DW_OP_regx	0x90	1	ULEB128 register
DW_OP_fbreg	0x91	1	SLEB128 offset
DW_OP_bregx	0x92	2	ULEB128 register followed by SLEB128 offset
DW_OP_piece	0x93	1	ULEB128 size of piece addressed
DW_OP_deref_size	0x94	1	1-byte size of data retrieved
DW_OP_xderef_size	0x95	1	1-byte size of data retrieved
DW_OP_nop	0x96	0	

Operation	Code	No. of Operands	Notes
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 followed by ULEB128 offset
DW_OP_implicit_value ‡	0x9e	2	ULEB128 size followed by block of that size
DW_OP_stack_value ‡	0x9f	0	
DW_OP_lo_user	0xe0		
DW_OP_hi_user	0xff		

‡ New in DWARF Version 4

Figure 24. DWARF operation encodings

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 selection entry, or an end of list entry.

A location list entry consists of two address offsets followed by a 2-byte length, followed by a block of contiguous bytes that contains a DWARF location description. The length specifies the

number of bytes in that block. The two offsets are the same size as an address on the target machine.

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

For a location list to be specified, the base address of the corresponding compilation unit must be defined (see Section 3.1.1).

7.8 Base Type Attribute Encodings

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

Base type encoding code 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
DW_ATE_unsigned_fixed	0x0e

Base type encoding code name	Value
DW_ATE_decimal_float	0x0f
DW_ATE_UTF ‡	0x10
DW_ATE_lo_user	0x80
DW_ATE_hi_user	0xff

‡ New in DWARF Version 4

Figure 25. Base type encoding values

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

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

Figure 26. Decimal sign encodings

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

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

Figure 27. Endianity encodings

7.9 Accessibility Codes

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

Accessibility code name	Value
DW_ACCESS_public	0x01
DW_ACCESS_protected	0x02
DW_ACCESS_private	0x03

Figure 28. Accessibility encodings

7.10 Visibility Codes

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

Visibility code name	Value
DW_VIS_local	0x01
DW_VIS_exported	0x02
DW_VIS_qualified	0x03

Figure 29. Visibility encodings

7.11 Virtuality Codes

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

Virtuality code name	Value
DW_VIRTUALITY_none	0x00
DW_VIRTUALITY_virtual	0x01
DW_VIRTUALITY_pure_virtual	0x02

Figure 30. Virtuality encodings

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 Figure 31. Names marked with † and their associated values are reserved, but the languages they represent are not well supported. Figure 31 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.

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

Language name	Value	Default Lower Bound
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_lo_user	0x8000	
DW_LANG_hi_user	0xffff	

† See text

Figure 31. Language encodings

7.13 Address Class Encodings

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

7.14 Identifier Case

The encodings of the constants used in the DW_AT_identifier_case attribute are given in Figure 32.

Identifier Case Name	Value
DW_ID_case_sensitive	0x00
DW_ID_up_case	0x01
DW_ID_down_case	0x02
DW_ID_case_insensitive	0x03

Figure 32. Identifier case encodings

7.15 Calling Convention Encodings

The encodings of the constants used in the DW_AT_calling_convention attribute are given in Figure 33.

Calling Convention Name	Value
DW_CC_normal	0x01
DW_CC_program	0x02
DW_CC_nocall	0x03
DW_CC_lo_user	0x40
DW_CC_hi_user	0xff

Figure 33. Calling convention encodings

7.16 Inline Codes

The encodings of the constants used in the DW_AT_inline attribute are given in Figure 34.

Inline Code Name	Value
DW_INL_not_inlined	0x00
DW_INL_inlined	0x01
DW_INL_declared_not_inlined	0x02
DW_INL_declared_inlined	0x03

Figure 34. Inline encodings

7.17 Array Ordering

The encodings of the constants used in the DW_AT_ordering attribute are given in Figure 35.

Ordering name	Value
DW_ORD_row_major	0x00
DW_ORD_col_major	0x01

Figure 35. Ordering encodings

7.18 Discriminant Lists

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

Descriptor Name	Value
DW_DSC_label	0x00
DW_DSC_range	0x01

Figure 36. Discriminant descriptor encodings

7.19 Name Lookup Tables

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

1. `unit_length` (initial length)

   A 4-byte or 12-byte length of the set of entries for this compilation unit, not including the length field itself. In the 32-bit DWARF format, this is a 4-byte unsigned integer (which must be less than `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).

2. `version` (uhalf)

   A 2-byte version identifier containing the value 2 (see Appendix F).

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 field is an 8-byte unsigned offset (see Section 7.4).

4. `debug_info_length` (section length)

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

This header is followed by a series of tuples. Each tuple consists of a 4-byte or 8-byte offset followed by a string of non-null bytes terminated by one null byte. In the 32-bit DWARF format, this is a 4-byte offset; in the 64-bit DWARF format, it is an 8-byte offset. Each set is terminated by an offset containing the value 0.

7.20 Address Range Table

Each set of entries in the table of address ranges contained in the `.debug_aranges` section begins with a header containing:

1. `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).

2. `version` (uhalf)

   A 2-byte version identifier containing the value 2 (see Appendix F).

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).

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_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 size is given by the `segment_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_size` field in the header is zero, the segment selectors are omitted from all tuples, including the terminating tuple.

7.21 Line Number Information

The version number in the line number program header is 4 (see Appendix F)..

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 Figure 37.

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

Opcode Name	Value
DW_LNS_set_epilogue_begin	0x0b
DW_LNS_set_isa	0x0c

Figure 37. Line Number Standard Opcode Encodings

The encodings for the extended opcodes are given in Figure 38.

Opcode Name	Value
DW_LNE_end_sequence	0x01
DW_LNE_set_address	0x02
DW_LNE_define_file	0x03
DW_LNE_set_discriminator ‡	0x04
DW_LNE_lo_user	0x80
DW_LNE_hi_user	0xff

‡ New in DWARF Version 4

Figure 38. Line Number Extended Opcode Encodings

7.22 Macro Information

The source line numbers and source file indices encoded in the macro information section are represented as unsigned LEB128 numbers as are the constants in a DW_MACINFO_vendor_ext entry.

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

Macinfo Type Name	Value
DW_MACINFO_define	0x01
DW_MACINFO_undef	0x02
DW_MACINFO_start_file	0x03
DW_MACINFO_end_file	0x04
DW_MACINFO_vendor_ext	0xff

Figure 39. Macinfo Type Encodings

7.23 Call Frame Information

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

The value of the CIE version number is 4 (see Appendix F).

Call frame instructions are encoded in one or more bytes. The primary opcode is encoded in the high order two bits of the first byte (that is, opcode = byte >> 6). An operand or extended opcode may be encoded in the low order 6 bits. Additional operands are encoded in subsequent bytes. The instructions and their encodings are presented in Figure 40.

Instruction	High 2 Bits	Low 6 Bits	Operand 1	Operand 2
DW_CFA_advance_loc	0x1	delta		
DW_CFA_offset	0x2	register	ULEB128 offset	
DW_CFA_restore	0x3	register		
DW_CFA_nop	0	0		
DW_CFA_set_loc	0	0x01	address	
DW_CFA_advance_loc1	0	0x02	1-byte delta	
DW_CFA_advance_loc2	0	0x03	2-byte delta	
DW_CFA_advance_loc4	0	0x04	4-byte delta	
DW_CFA_offset_extended	0	0x05	ULEB128 register	ULEB128 offset
DW_CFA_restore_extended	0	0x06	ULEB128 register	
DW_CFA_undefined	0	0x07	ULEB128 register	
DW_CFA_same_value	0	0x08	ULEB128 register	
DW_CFA_register	0	0x09	ULEB128 register	ULEB128 register
DW_CFA_remember_state	0	0x0a		
DW_CFA_restore_state	0	0x0b		
DW_CFA_def_cfa	0	0x0c	ULEB128 register	ULEB128 offset
DW_CFA_def_cfa_register	0	0x0d	ULEB128 register	
DW_CFA_def_cfa_offset	0	0x0e	ULEB128 offset	
DW_CFA_def_cfa_expression	0	0x0f	BLOCK	
DW_CFA_expression	0	0x10	ULEB128 register	BLOCK

Instruction	High 2 Bits	Low 6 Bits	Operand 1	Operand 2
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		

Figure 40. Call frame instruction encodings

7.24 Non-contiguous Address Ranges

Each entry in a range list (see Section 2.17.3) is either a range list entry, a base address selection entry, or an end of list entry.

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

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

For a range list to be specified, the base address of the corresponding compilation unit must be defined (see Section 3.1.1).

7.25 Dependencies and Constraints

The debugging information in this format is intended to exist in the `.debug_abbrev`, `.debug_aranges`, `.debug_frame`, `.debug_info`, `.debug_line`, `.debug_loc`, `.debug_macinfo`, `.debug_pubnames`, `.debug_pubtypes`, `.debug_ranges`, `.debug_str` and `.debug_types` sections of an object file, or equivalent separate file or database. The information is not word-aligned. Consequently:

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

7.26 Integer Representation Names

The sizes of the integers used in the lookup by name, lookup by address, line number and call frame information sections are given in Figure 41.

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

Figure 41. Integer Representation Names

7.27 Type Signature Computation

A type signature is computed only by the DWARF producer; it is used by a DWARF consumer to resolve type references to the type definitions that are contained in type units.

The type signature for a type T0 is formed from the MD5 hash 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.
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 following attributes that are present in the debugging information entry, in the order listed below, append to S a marker letter (see below), the DWARF attribute code, and the attribute value.

    DW_AT_name

    DW_AT_accessibility

    DW_AT_address_class

    DW_AT_allocated

    DW_AT_artificial

    DW_AT_associated

    DW_AT_binary_scale

    DW_AT_bit_offset

    DW_AT_bit_size

    DW_AT_bit_stride

    DW_AT_byte_size

    DW_AT_byte_stride

    DW_AT_const_expr

    DW_AT_const_value

    DW_AT_containing_type

    DW_AT_count

    DW_AT_data_bit_offset

    DW_AT_data_location

    DW_AT_data_member_location

    DW_AT_decimal_scale

    DW_AT_decimal_sign

    DW_AT_default_value

    DW_AT_digit_count

    DW_AT_discr

    DW_AT_discr_list

    DW_AT_discr_value

DW_AT_encoding
DW_AT_enum_class
DW_AT_endianity
DW_AT_explicit
DW_AT_is_optional
DW_AT_location
DW_AT_lower_bound
DW_AT_mutable
DW_AT_ordering
DW_AT_picture_string
DW_AT_prototyped
DW_AT_small
DW_AT_segment
DW_AT_string_length
DW_AT_threads_scaled
DW_AT_upper_bound
DW_AT_use_location
DW_AT_use_UTF8
DW_AT_variable_parameter
DW_AT_virtuality
DW_AT_visibility
DW_AT_vtable_elem_location

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 should also be 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; otherwise, (b) 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, 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 name to be used is the ABI-specific name of the subprogram (e.g., 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.

An attribute that refers to another type entry should be 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 should not be considered 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 should not be considered for a separate type unit.

A type should be considered for a separate type unit only if all of the type entries that it contains or refers to in Steps 6 and 7 can themselves each be considered 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 MD5 hash is computed for the string and the lower 64 bits are taken as the type signature.

The string S is intended to be a flattened representation of the type that uniquely identifies that type (i.e., a different type is highly unlikely to produce the same string).

A debugging information entry should not be placed in a separate type unit if any of the following apply:

- *The entry has an attribute whose value is a location expression, and the location expression contains a reference to another debugging information entry (e.g., 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 hash.*

- *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 DIE 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 hash may be found in Appendix E.2.

Appendix A -- Attributes by Tag Value (informative)

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

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

Figure 42, Attributes by tag value, begins here.

TAG Name	Applicable Attributes
DW_TAG_access_declaration	DECL DW_AT_accessibility DW_AT_description DW_AT_name DW_AT_sibling
DW_TAG_array_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_allocated DW_AT_associated DW_AT_bit_size DW_AT_bit_stride DW_AT_byte_size DW_AT_data_location DW_AT_declaration DW_AT_description DW_AT_name DW_AT_ordering DW_AT_sibling DW_AT_specification DW_AT_start_scope DW_AT_type DW_AT_visibility

DW_TAG_base_type	DECL DW_AT_allocated DW_AT_associated DW_AT_binary_scale DW_AT_bit_offset 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_description DW_AT_digit_count DW_AT_encoding DW_AT_endianity DW_AT_name DW_AT_picture_string DW_AT_sibling DW_AT_small
DW_TAG_catch_block	DECL DW_AT_abstract_origin DW_AT_high_pc DW_AT_low_pc DW_AT_ranges DW_AT_segment DW_AT_sibling

DW_TAG_class_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_allocated DW_AT_associated DW_AT_bit_size DW_AT_byte_size DW_AT_data_location DW_AT_declaration DW_AT_description DW_AT_name DW_AT_sibling DW_AT_signature DW_AT_specification DW_AT_start_scope DW_AT_visibility
DW_TAG_common_block	DECL DW_AT_declaration DW_AT_description DW_AT_linkage_name DW_AT_location DW_AT_name DW_AT_segment DW_AT_sibling DW_AT_visibility
DW_TAG_common_inclusion	DECL DW_AT_common_reference DW_AT_declaration DW_AT_sibling DW_AT_visibility

DW_TAG_compile_unit	DW_AT_base_types DW_AT_comp_dir DW_AT_identifier_case DW_AT_high_pc DW_AT_language DW_AT_low_pc DW_AT_macro_info DW_AT_main_subprogram DW_AT_name DW_AT_producer DW_AT_ranges DW_AT_segment DW_AT_stmt_list DW_AT_use_UTF8
DW_TAG_condition	DECL DW_AT_name DW_AT_sibling
DW_TAG_const_type	DW_AT_allocated DW_AT_associated DW_AT_data_location DW_AT_name DW_AT_sibling DW_AT_type

DW_TAG_constant	DECL DW_AT_accessibility DW_AT_const_value DW_AT_declaration DW_AT_description DW_AT_endianity DW_AT_external DW_AT_linkage_name DW_AT_name DW_AT_sibling DW_AT_start_scope DW_AT_type DW_AT_visibility
DW_TAG_dwarf_procedure	DW_AT_location
DW_TAG_entry_point	DECL DW_AT_address_class DW_AT_description DW_AT_frame_base DW_AT_linkage_name DW_AT_low_pc DW_AT_name DW_AT_return_addr DW_AT_segment DW_AT_sibling DW_AT_static_link DW_AT_type

DW_TAG_enumeration_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_allocated DW_AT_associated DW_AT_bit_size DW_AT_bit_stride DW_AT_byte_size DW_AT_byte_stride DW_AT_data_location DW_AT_declaration DW_AT_description DW_AT_enum_class DW_AT_name DW_AT_sibling 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_description DW_AT_name DW_AT_sibling

DW_TAG_file_type	DECL DW_AT_abstract_origin DW_AT_allocated DW_AT_associated DW_AT_bit_size DW_AT_byte_size DW_AT_data_location DW_AT_description DW_AT_name DW_AT_sibling DW_AT_start_scope DW_AT_type DW_AT_visibility
DW_TAG_formal_parameter	DECL DW_AT_abstract_origin DW_AT_artificial DW_AT_const_value DW_AT_default_value DW_AT_description DW_AT_endianity DW_AT_is_optional DW_AT_location DW_AT_name DW_AT_segment DW_AT_sibling DW_AT_type DW_AT_variable_parameter
DW_TAG_friend	DECL DW_AT_abstract_origin DW_AT_friend DW_AT_sibling

DW_TAG_imported_declaration	DECL DW_AT_accessibility DW_AT_description DW_AT_import DW_AT_name DW_AT_sibling DW_AT_start_scope
DW_TAG_imported_module	DECL DW_AT_import DW_AT_sibling DW_AT_start_scope
DW_TAG_imported_unit	DW_AT_import
DW_TAG_inheritance	DECL DW_AT_accessibility DW_AT_data_member_location DW_AT_sibling DW_AT_type DW_AT_virtuality
DW_TAG_inlined_subroutine	DW_AT_abstract_origin 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_sibling DW_AT_start_scope DW_AT_trampoline

DW_TAG_interface_type	DECL DW_AT_accessibility DW_AT_description DW_AT_name DW_AT_sibling DW_AT_start_scope
DW_TAG_label	DECL DW_AT_abstract_origin DW_AT_description DW_AT_low_pc DW_AT_name DW_AT_segment DW_AT_start_scope DW_AT_sibling
DW_TAG_lexical_block	DECL DW_AT_abstract_origin DW_AT_description DW_AT_high_pc DW_AT_low_pc DW_AT_name DW_AT_ranges DW_AT_segment DW_AT_sibling

DW_TAG_member	DECL DW_AT_accessibility DW_AT_bit_offset DW_AT_bit_size DW_AT_byte_size DW_AT_data_bit_offset DW_AT_data_member_location DW_AT_declaration DW_AT_description DW_AT_mutable DW_AT_name DW_AT_sibling DW_AT_type DW_AT_visibility
DW_TAG_module	DECL DW_AT_accessibility DW_AT_declaration DW_AT_description DW_AT_entry_pc DW_AT_high_pc DW_AT_low_pc DW_AT_name DW_AT_priority DW_AT_ranges DW_AT_segment DW_AT_sibling DW_AT_specification DW_AT_visibility
DW_TAG_namelist	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_declaration DW_AT_name DW_AT_sibling DW_AT_visibility

DW_TAG_namelist_item	DECL DW_AT_namelist_item DW_AT_sibling
DW_TAG_namespace	DECL DW_AT_description DW_AT_extension DW_AT_name DW_AT_sibling DW_AT_start_scope
DW_TAG_packed_type	DW_AT_allocated DW_AT_associated DW_AT_data_location DW_AT_name DW_AT_sibling DW_AT_type
DW_TAG_partial_unit	DW_AT_base_types DW_AT_comp_dir DW_AT_description DW_AT_identifier_case DW_AT_high_pc DW_AT_language DW_AT_low_pc DW_AT_macro_info DW_AT_main_subprogram DW_AT_name DW_AT_producer DW_AT_ranges DW_AT_segment DW_AT_stmt_list DW_AT_use_UTF8

DW_TAG_pointer_type	DW_AT_address_class DW_AT_allocated DW_AT_associated DW_AT_data_location DW_AT_name DW_AT_sibling DW_AT_type
DW_TAG_ptr_to_member_type	DECL DW_AT_abstract_origin DW_AT_address_class DW_AT_allocated DW_AT_associated DW_AT_containing_type DW_AT_data_location DW_AT_declaration DW_AT_description DW_AT_name DW_AT_sibling DW_AT_type DW_AT_use_location DW_AT_visibility
DW_TAG_reference_type	DW_AT_address_class DW_AT_allocated DW_AT_associated DW_AT_data_location DW_AT_name DW_AT_sibling DW_AT_type
DW_TAG_restrict_type	DW_AT_allocated DW_AT_associated DW_AT_data_location DW_AT_name DW_AT_sibling DW_AT_type

DW_TAG_rvalue_reference_type	DECL DW_AT_address_class DW_AT_allocated DW_AT_associated DW_AT_data_location DW_AT_name DW_AT_sibling DW_AT_type
DW_TAG_set_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_allocated DW_AT_associated DW_AT_bit_size DW_AT_byte_size DW_AT_data_location DW_AT_declaration DW_AT_description DW_AT_name DW_AT_start_scope DW_AT_sibling DW_AT_type DW_AT_visibility
DW_TAG_shared_type	DW_AT_allocated DW_AT_associated DW_AT_count DW_AT_data_location DW_AT_name DW_AT_sibling DW_AT_type

DW_TAG_string_type	DECL DW_AT_abstract_origin 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_description DW_AT_name DW_AT_sibling DW_AT_start_scope DW_AT_string_length DW_AT_visibility
DW_TAG_structure_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_allocated DW_AT_associated DW_AT_bit_size DW_AT_byte_size DW_AT_data_location DW_AT_declaration DW_AT_description DW_AT_name DW_AT_sibling DW_AT_signature DW_AT_specification DW_AT_start_scope DW_AT_visibility

DW_TAG_subprogram	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_address_class DW_AT_artificial DW_AT_calling_convention DW_AT_declaration DW_AT_description DW_AT_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_object_pointer DW_AT_prototyped DW_AT_pure DW_AT_ranges DW_AT_recursive DW_AT_return_addr DW_AT_segment DW_AT_sibling DW_AT_specification DW_AT_start_scope DW_AT_static_link DW_AT_trampoline DW_AT_type DW_AT_visibility DW_AT_virtuality DW_AT_vtable_elem_location

DW_TAG_subrange_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_allocated DW_AT_associated DW_AT_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_description DW_AT_lower_bound DW_AT_name DW_AT_sibling DW_AT_threads_scaled DW_AT_type DW_AT_upper_bound DW_AT_visibility
DW_TAG_subroutine_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_address_class DW_AT_allocated DW_AT_associated DW_AT_data_location DW_AT_declaration DW_AT_description DW_AT_name DW_AT_prototyped DW_AT_sibling DW_AT_start_scope DW_AT_type DW_AT_visibility

DW_TAG_template_alias	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_allocated DW_AT_associated DW_AT_data_location DW_AT_declaration DW_AT_description DW_AT_name DW_AT_sibling DW_AT_signature DW_AT_start_scope DW_AT_type DW_AT_visibility
DW_TAG_template_type_parameter	DECL DW_AT_description DW_AT_name DW_AT_sibling DW_AT_type
DW_TAG_template_value_parameter	DECL DW_AT_const_value DW_AT_description DW_AT_name DW_AT_sibling DW_AT_type
DW_TAG_thrown_type	DECL DW_AT_allocated DW_AT_associated DW_AT_data_location DW_AT_sibling DW_AT_type

DW_TAG_try_block	DECL DW_AT_abstract_origin DW_AT_high_pc DW_AT_low_pc DW_AT_ranges DW_AT_segment DW_AT_sibling
DW_TAG_typedef	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_allocated DW_AT_associated DW_AT_data_location DW_AT_declaration DW_AT_description DW_AT_name DW_AT_sibling DW_AT_start_scope DW_AT_type DW_AT_visibility
DW_TAG_type_unit	DW_AT_language

DW_TAG_union_type	DECL DW_AT_abstract_origin 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_description DW_AT_name DW_AT_sibling DW_AT_signature DW_AT_specification DW_AT_start_scope DW_AT_visibility
DW_TAG_unspecified_parameters	DECL DW_AT_abstract_origin DW_AT_artificial DW_AT_sibling
DW_TAG_unspecified_type	DECL DW_AT_description DW_AT_name

DW_TAG_variable	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_const_expr DW_AT_const_value DW_AT_declaration DW_AT_description DW_AT_endianity DW_AT_external DW_AT_linkage_name DW_AT_location DW_AT_name DW_AT_segment DW_AT_sibling DW_AT_specification DW_AT_start_scope DW_AT_type DW_AT_visibility
DW_TAG_variant	DECL DW_AT_accessibility DW_AT_abstract_origin DW_AT_declaration DW_AT_discr_list DW_AT_discr_value DW_AT_sibling
DW_TAG_variant_part	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_declaration DW_AT_discr DW_AT_sibling DW_AT_type

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

Figure 42. Attributes by TAG value

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 diagram and associated notes on the following pages.

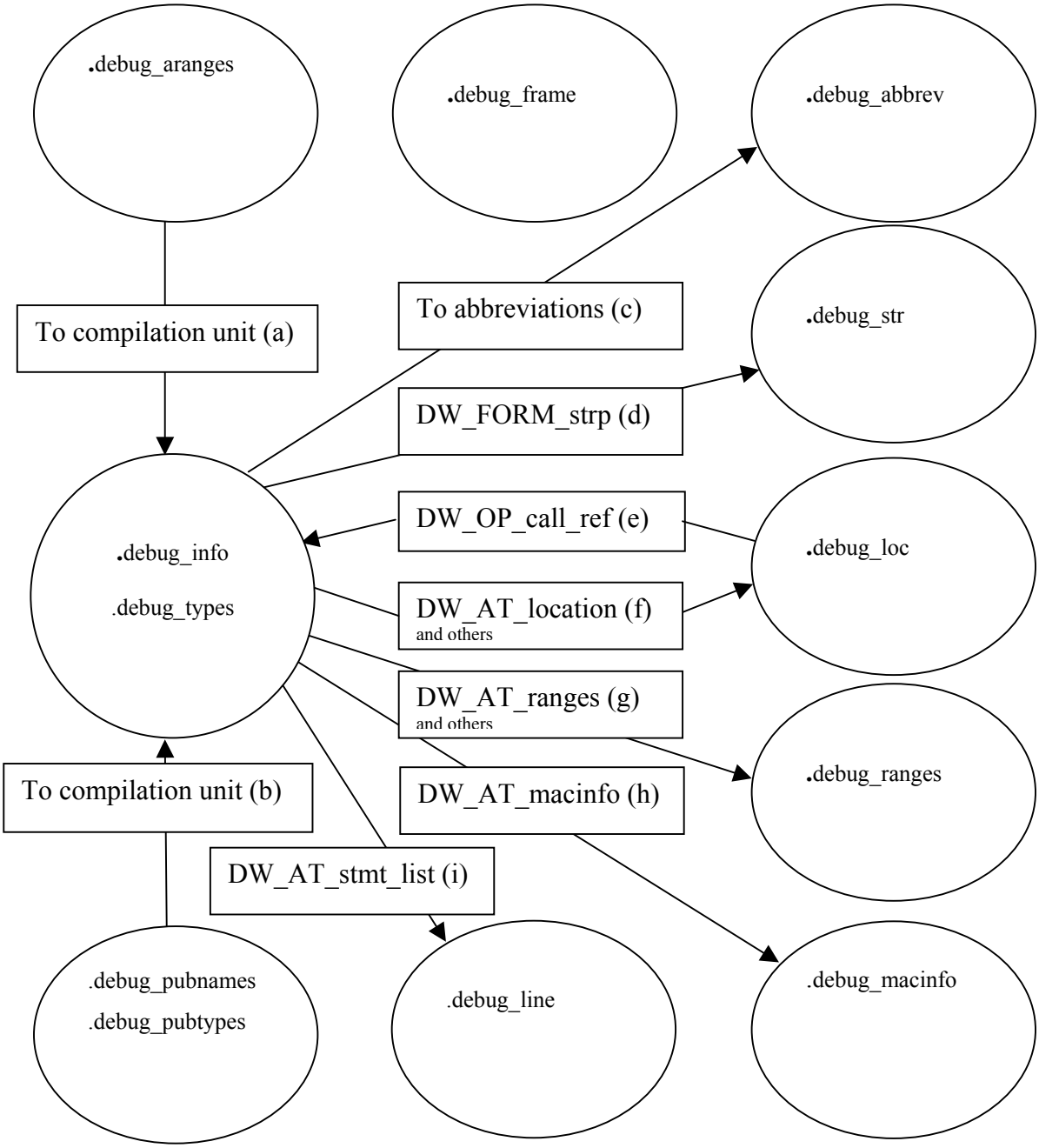


Figure 43. Debug section relationships

Notes

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

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

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

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

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

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

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

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

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.

```
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 44. 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 45. 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 46. Algorithm to decode an unsigned LEB128 number

```
result = 0;
shift = 0;
size = number of bits in signed integer;
while(true)
{
     byte = next byte in input;
     result |= (low order 7 bits of byte << shift);
     shift += 7;
     /* sign bit of byte is 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 47. Algorithm to decode a signed LEB128 number

Appendix D -- Examples (informative)

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

D.1 Compilation Units and Abbreviations Table Example

Figure 48 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: Version 1.3” “/home/mydir/src” DW_LANG_C89 0x0 0x55 DW_FORM_sec_offset 0x0
e1:	2 “char” DW_ATE_unsigned_char 1
e2:	3 *e1*
	4 “POINTER” *e2*
	0

Compilation Unit 2: `.debug_info`

length *4* *a1 (abbreviations table offset)* 4
...
4 “strp” *e2*
...

Abbreviation Table - `.debug_abbrev`

a1:	1 DW_TAG_compile_unit DW_CHILDREN_yes	
	DW_AT_name	DW_FORM_string
	DW_AT_producer	DW_FORM_string
	DW_AT_comp_dir	DW_FORM_string
	DW_AT_language	DW_FORM_data1
	DW_AT_low_pc	DW_FORM_addr
	DW_AT_high_pc	DW_FORM_data1
	DW_AT_stmt_list	DW_FORM_indirect
	0	0
	2 DW_TAG_base_type DW_CHILDREN_no	
	DW_AT_name	DW_FORM_string
	DW_AT_encoding	DW_FORM_data1
	DW_AT_byte_size	DW_FORM_data1
	0	0
	3 DW_TAG_pointer_type DW_CHILDREN_no	
	DW_AT_type	DW_FORM_ref4
	0	0
	4 DW_TAG_typedef DW_CHILDREN_no	
	DW_AT_name	DW_FORM_string
	DW_AT_type	DW_FORM_ref_addr
	0	0
	0	

Figure 48. Compilation units and abbreviations table

D.2 Aggregate Examples

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

D.2.1 Fortran 90 Example

Consider the Fortran 90 source fragment in Figure 49.

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

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

allocate(arrays(20))

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

Figure 49. Fortran 90 example: source fragment

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

For concreteness, suppose that a descriptor looks something like the C structure in Figure 50. 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 50. Fortran 90 example: descriptor representation

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

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

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

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

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

Similarly for variable `arrays`:

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

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

Finally, the following notation is useful:

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

The DWARF description is shown in Figure 51.

```
! 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<x> where
                !    x == 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).

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

Figure 51. Fortran 90 example: DWARF description

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

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

Interpretation of this expression proceeds as follows:

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

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

   a) the address of the array data

   b) the lower bounds of the array

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

   c) the stride

   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 `arrays`, whose address was found in step 1. (Had there been no DW_AT_data_location attribute, the desired address would be the same as the address from step 1.)

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

For c), the default stride 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$).

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

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

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

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

4) Find the 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'll 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 Ada Example

Figure 52 illustrates two kinds of Ada parameterized array, one embedded in a record.

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

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

OBJ2B : REC2;
```

Figure 52. Ada example: source fragment

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

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

The DWARF description is shown in Figure 53.

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 what should be the parent type. For this reason, DWARF for the unconstrained array ARR is not needed for the purposes of this example and therefore not shown.

```
11$:    DW_TAG_variable
            DW_AT_name("M")
            DW_AT_type(reference to 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 53. Ada example: DWARF description

D.2.3 Pascal Example

The Pascal source in Figure 54 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 54. Packed record example: source fragment

The DWARF representation in Figure 55 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 5.5.6.

```
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)
            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_AT_member
                DW_AT_name("F2")
                DW_AT_type(reference to 21$)
                DW_AT_data_bit_offset(1)
                DW_AT_bit_size(32)      ! may be omitted
            DW_AT_member
                DW_AT_name("F4")
                DW_AT_type(reference to 22$)
                DW_AT_data_bit_offset(33)
                DW_AT_bit_size(4)       ! may be omitted
            DW_AT_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 55. Packed record example: DWARF description

D.3 Namespace Example

The C++ example in Figure 56 is used to illustrate the representation of namespaces.

```
namespace {
    int i;
}

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

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

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

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

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

using namespace Foo;              // (5) using directive

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

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

Figure 56. Namespace example: source fragment

The DWARF representation in Figure 57 is appropriate.

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

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

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

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

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

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

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

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

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

Figure 57. Namespace example: DWARF description

D.4 Member Function Example

Consider the member function example fragment in Figure 58.

```
class A
{
     void func1(int x1);
     void func2() const;
     static void func3(int x3);
};

void A::func1(int x) {}
```

Figure 58. Member function example: source fragment

The DWARF description in Figure 59 is appropriate.

```
1$:     DW_TAG_unspecified_type
            DW_AT_name("void")
            ...
2$      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$:         DW_TAG_subprogram
                DW_AT_declaration
                DW_AT_name("func1")
                DW_AT_type(reference to 1$)
                DW_AT_object_pointer(reference to 8$)
                    ! References a formal parameter in this member function
                ...
8$:             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_type(reference to 1$)
                DW_AT_object_pointer(reference to 11$)
                    ! 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$:        DW_TAG_subprogram
                DW_AT_declaration
                DW_AT_name("func3")
                DW_AT_type(reference to 1$)
                ...
                    ! 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 59. Member function example: DWARF description

D.5 Line Number Program Example

Consider the simple source file and the resulting machine code for the Intel 8086 processor in Figure 60.

```
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 60. Line number program example: machine code

Suppose the line number program header includes the following (header fields not needed below are not shown):

```
version                      4
minimum_instruction_length   1
opcode_base                 10    ! Opcodes 10-12 not needed
line_base                    1
line_range                  15
```

Figure 61 shows one encoding of the line number program, which occupies 12 bytes (the opcode SPECIAL(*m,n*) indicates the special opcode generated for a line increment of *m* and an address increment of *n*).

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

Figure 61. Line number program example: one encoding

Figure 62 shows an alternate encoding of the same program using standard opcodes to advance the program counter; this encoding occupies 22 bytes.

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

Figure 62. Line number program example: alternate encoding

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 8 4-byte registers:

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

- The stack grows in the negative direction.
- The architectural ABI committee specifies that the stack pointer (R7) is the same as the CFA

The following are two code fragments from a subroutine called foo that uses a frame pointer (in addition to the stack pointer). The first column values are byte addresses. <fs> denotes the stack frame size in bytes, namely 12.

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

Figure 63. Call frame information example: machine code fragments

An abstract table (see Section 6.4.1) for the foo subroutine is shown in Figure 64. Corresponding fragments from the `.debug_frame` section are shown in Figure 65.

The following notations apply in Figure 64:

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

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

Figure 64. Call frame information example: conceptual matrix

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		

Figure 65. Call frame information example: common information entry encoding

The following notations apply in Figure 66:

1. <fs> = frame size
2. <caf> = code alignment factor
3. <daf> = data alignment factor

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> (second 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		

Figure 66. Call frame information example: frame description entry encoding

D.7 Inlining Examples

The pseudo-source in Figure 67 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 67. Inlining examples: pseudo-source fragment

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

The approaches are:

1. Inline both OUTER and INNER in all cases

2. Inline OUTER, multiple INNERs

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

3. Inline OUTER, one INNER

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

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

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

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

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

For conciseness, declaration coordinates and call coordinates are omitted.

D.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 68.

Notice in Figure 68 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 67 might be described as shown in Figure 69.

```
        ! 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 68. 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 69. Inlining example #1: concrete instance

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 70.

Note that the debugging information in this Figure differs from that in Figure 68 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 70 by the fact that the labels for INNER use the substring OUTER instead of INNER.

A resulting concrete inlined instance of OUTER is shown in Figure 71.

Notice in Figure 71 that OUTER is expanded as a concrete inlined instance, and that INNER is nested within it as a concrete out-of-line subprogram. Because INNER is cloned for each inline expansion of OUTER, only the invariant attributes of INNER (for example, DW_AT_name) are specified in the abstract instance of OUTER, and the low-level, instance-specific attributes of INNER (for example, DW_AT_low_pc) are specified in each concrete instance of OUTER.

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

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

Figure 70. Inlining example #2: abstract instance

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

Figure 71. Inlining example #2: concrete instance

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 72.

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 73.

Notice in Figure 73 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 up-level variables, or a compiler generated static link like parameter that points to the group of up-level entities, among other possibilities. In either of these cases, the DWARF description for the location attribute of each uplevel variable needs to be different if accessed from within INNER compared to when accessed from within the instances of OUTER. An implementation is likely to require vendor-specific DWARF attributes and/or debugging information entries to describe such cases.

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

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

Figure 72. 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 73. 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.

```
constexpr double mass = 9.8;

constexpr int square (int x) { return x * x; }

float arr[square(9)];       // square() called and inlined
```

Figure 74. Constant expressions: C++ source

These declarations can be represented as illustrated in Figure 75.

```
      ! 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 75. Constant expressions: DWARF description

D.9 Unicode Character Example

Unicode character encodings can be described in DWARF as illustrated in Figure 76.

```
// C++ source
//
char16_t chr_a = u'h';
char32_t chr_b = U'h';

! 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 76. Unicode character type examples

D.10 Type-Safe Enumeration Example

C++ type-safe enumerations can be described in DWARF as illustrated in Figure 77.

```
// C++ source
//
enum class E { E1, E2=100 };

E e1;


! 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 77. C++ type-safe enumeration example

D.11 Template Example

C++ templates can be described in DWARF as illustrated in Figure 78.

```
// C++ source
//
   template<class T>
   struct wrapper {
      T comp;
   };
   wrapper<int>  obj;


! 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 78. C++ template example #1

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.

There exist situations where it is not possible for the DWARF to imply anything about the nature of the original template. Consider the C++ source in Figure 79.

```
// C++ source
//
   template<class T>
   struct wrapper {
      T comp;
   };
   template<class U>
   void consume(wrapper<U>  formal)
   {
      ...
   }
   wrapper<int>  obj;
   consume(obj);


! 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 79. C++ template example #2

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 of `consume` were declared in the source to be `wrapper<int>`.

D.12 Template Alias Examples

C++ template aliases can be described in DWARF as illustrated in Figure 80 and Figure 81

```
// C++ source
//
template<typename T, typename U>
struct Alpha {
   T tango;
   U uniform;
};

template<typename V> using Beta = Alpha<V,V>;

Beta<long> b;


! 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 80. Template alias example #1

```
// C++ source
//
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;


! DWARF representation for X<Y<int>>
!
30$:  DW_TAG_structure_type                          // struct Y<int>
          DW_AT_name("Y")
31$:      DW_TAG_template_type_parameter
              DW_AT_name("TY")
              DW_AT_type(reference to "int")
32$:  DW_TAG_structure_type                          // struct X<Y<int>>
          DW_AT_name("X")
33$:      DW_TAG_template_type_parameter
              DW_AT_name("TX")
              DW_AT_type(reference to 30$)

! DWARF representation for X<Z<int>>
!
40$:  DW_TAG_template_alias                          // template<class T>
using Z = Y<int>;
          DW_AT_name("Z")
          DW_AT_type(reference to 30$)
41$:      DW_TAG_template_type_parameter
              DW_AT_name("T")
              DW_AT_type(reference to "int")
42$:  DW_TAG_structure_type                          // struct X<Z<int>>
          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 81. Template alias example #2

APPENDIX D–EXAMPLES (INFORMATIVE)

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 from a single compilation might contain sections named:

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

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

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

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

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

==== ...

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

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

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

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 of referencing from inside one `.debug_info` compilation unit to another `.debug_info` compilation unit (DW_FORM_ref_addr provides this).
2. A means of having multiple contributions to specific sections (for example, `.debug_info`, and so on) in a single object file.
3. A means of identifying a section group (giving it a name).
4. A means of identifying which sections go together to make up a section group, so that the group can be treated as a unit (kept or discarded).
5. A means of indicating how each section group should be processed by the linker.

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

E.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 82 through Figure 89 of Section E.1.3.

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 (need explained below) within a section group can be given names of the form

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

such as

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

where

- `<prefix>` distinguishes this as a DWARF debug info name, and should identify the producer and, when appropriate, the language.
- `<file-designator>` and `<gid-number>` are as above.
- `<die-number>` could be a number sequentially assigned to entities (tokens, perhaps) found during compilation.

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

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

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

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

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

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

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

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

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 procedure or module context.

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

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

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

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.

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 figures compact and easier to read.

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

C++ Example

The C++ source in Figure 82 is used to illustrate the DWARF representation intended to allow duplicate elimination.

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

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

Figure 82. Duplicate elimination example #1: C++ source

Figure 83 shows the section group corresponding to the included file `wa.h`.

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

== section .debug_info

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

Figure 83. Duplicate elimination example #1: DWARF section group

Figure 84 shows the “normal” DWARF sections, which are not part of any section group, and how they make use of the information in the section group shown above.

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

== section .debug_info

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

Figure 84. Duplicate elimination example #1: primary compilation unit

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

Fortran Example

For a Fortran example, consider Figure 85.

```
---- 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 85. Duplicate elimination example #2: Fortran source

Figure 86 shows the section group corresponding to the included file `CommonStuff.fh`.

```
==== Section group name:

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

== section .debug_info

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

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

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

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

Figure 86. Duplicate elimination example #2: DWARF section group

Figure 87 shows the sections for the primary compilation unit.

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

== section .debug_info

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

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

Figure 87. Duplicate elimination example #2: primary unit

A companion main program is shown in Figure 88.

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

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

Figure 88. 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 89.

```
== 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 89. Duplicate elimination example #2: companion DWARF

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

C Example

The C++ example in this Section might appear to be equally valid as a C example. However, it is prudent to include a DW_TAG_imported_unit in the primary unit (see Figure 84) with an DW_AT_import attribute that refers to the proper unit in the section group.

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

E.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 `.debug_types` section and arranging for the linker to recognize and 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 these additional sections:

```
.debug_types
```

As discussed in the previous section (Section E.1), 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 `.debug_types` sections, the key is the type signature formed from the algorithm given in Section 7.27.

E.2.1 Signature Computation Example

As an example, consider a C++ header file containing the type definitions shown in Figure 90.

```
 1  namespace N {
 2
 3  struct B;
 4
 5  struct C {
 6      int x;
 7      int y;
 8  };
 9
10  class A {
11    public:
12      A(int v);
13      int v();
14    private:
15      int v_;
16      struct A *next;
17      struct B *bp;
18      struct C c;
19  };
20
21  }
```

Figure 90. Type signature examples: C++ source

Next, consider one possible representation of the DWARF information that describes the type "struct C" as shown in Figure 91:

```
  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 91. Type signature computation #1: DWARF representation

In computing a signature for the type N::C, flatten the type description into a byte stream according to the procedure outlined in Section 7.27. The result is shown in Figure 92.

```
// 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 0x78 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 92. Type signature computation #1: flattened byte stream

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 93.

```
  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
```

```
        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 93. Type signature computation #2: DWARF representation

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_sig8 to refer to the type signature for that type.

In computing a signature for the type N::A, flatten the type description into a byte stream according to the procedure outlined in Section 7.27. The result is shown in Figure 1.

Figure 94, Type signature example #2: flattened byte stream, begins here.

```
// 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
    // 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_AT_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_AT_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
    // 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 94. Type signature example #2: flattened byte stream

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 that shown in Figure 95.

```
DW_TAG_compile_unit
  ...
  DW_TAG_subprogram
    ...
    DW_TAG_variable
      DW_AT_name: "a"
      DW_AT_type: (signature) 0xd6d160f5 5589f6e9
      DW_AT_location: ...
  ...
```

Figure 95. Type signature example usage

E.2.2 Type Signature Computation Grammar

Figure 96 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 7.27.

```
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 96. Type signature computation grammar

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 objects. The linker wants to keep only one of a given entity. The DWARF description, and everything else for this function, should be reduced to just a single copy.

For each such code group (function template in this example) the compiler assigns a name for the group which will match all other instantiations of this function but match nothing else. The 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 might be slightly different in that the <file-designator> should be interpreted as a <function-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.

Appendix F – 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. Figure 97 shows what version numbers are in use for each section.

There are sections with no version number encoded in them; they are only accessed via the `.debug_info` and `.debug_types` sections and so an incompatible change in those sections' format would be represented by a change in the `.debug_info` and `.debug_types` section version number.

Section Name	Section version number in DWARF Version 2 (July 1993)	Section version number in DWARF Version 3 (December 2005)	Section version number in DWARF Version 4 (this document)
`.debug_abbrev`	-	-	-
`.debug_aranges`	2	2	2
`.debug_frame`	1	3	4
`.debug_info`	2	3	4
`.debug_line`	2	3	4
`.debug_loc`	-	-	-
`.debug_macinfo`	-	-	-
`.debug_pubnames`	2	2	2
`.debug_pubtypes`	x	2	2
`.debug_ranges`	x	-	-
`.debug_str`	-	-	-
`.debug_types`	x	x	4

Figure 97. Section version numbers

Notes:

- "-" means that a version number is not applicable (the section's header does not include a version).
- "x" means that the section was not defined in that version of the DWARF standard.
- The version numbers for the .debug_info and .debug_types sections must be the same.

For .debug_frame, section version 2 is unused.

Higher numbers are reserved for future use.

GNU Free Documentation License

Version 1.3, 3 November 2008

0. 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.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work.

Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent.

An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for

which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition.

Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material.

If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You

may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section.

   You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice.

These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 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".

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.

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 is not used to limit the legal rights of the compilation's users beyond what the individual works permit.

When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form.

Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

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.

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.

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 principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
>
> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
>
> A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

... parameters.....*See* unspecified parameters entry
.data 264
.debug_abbrev..... 141, 144, 145, 183, 215, 264, 276, 289
  `example` 219
.debug_aranges 107, 140, 141, 177, 183, 215, 264, 289
.debug_frame 128, 130, 140, 141, 183, 289
  `example` 240
.debug_info ...7, 24, 105, 106, 107, 108, 140, 141, 142, 143, 145, 146, 149, 151, 176, 177, 183, 215, 264, 265, 267, 268, 269, 271, 273, 274, 275, 276, 287, 289
  `example` 219
.debug_line 45, 108, 140, 141, 148, 183, 215, 264, 276, 289
.debug_loc.....30, 148, 183, 215, 289
.debug_macinfo 45, 123, 125, 149, 183, 215, 289
.debug_pubnames ... 106, 140, 141, 142, 176, 183, 215, 289
.debug_pubtypes 106, 140, 141, 142, 176, 183, 215, 289
.debug_ranges 38, 149, 183, 215, 289
.debug_str 142, 150, 151, 183, 215, 289
.debug_types .7, 24, 105, 107, 140, 141, 142, 143, 144, 145, 146, 183, 276, 289
.text 264, 271, 274
<caf>.....*See* code alignment factor
<daf> .....*See* data alignment factor
32-bit DWARF format.....140
64-bit DWARF format.....140
abbreviations table 143, 145
  dynamic forms in.....146
  example 219
abstract instance.....288
  example 245, 248, 251
  nested.....63
abstract instance entry.....59
abstract instance root 59
abstract instance tree.....59, 61
abstract origin attribute 61, 63
accelerated access 105
  by address.....107
  by name 106
access declaration entry 87
accessibility attribute 32, 87, 88, 92
  encoding 170
activation, call frame.....126
Ada 1, 9, 32, 41, 44, 79, 80, 81, 103, 222, 227, 228, 229
address
  dereference operator 19, 20
  implicit push for member pointer..... 101
  implicit push of base 20
  size of an .....*See* size of an address
address class..... 15, 147
address class attribute..... 34, 55, 81
  encoding..... *173*
address range
  in location list..... 31
  in range list..... 39
address register
  in call frame information..... 127
  in line number machine..... 109
address selection.....*See* base address selection
address size .....*See* size of an address
address space
  flat34
  multiple 19, 20
  segmented 34, 107, 144, 177
address, uplevel ..... *See* static link attribute
address_size 107, 129, 144, 177, 178
alias declaration.....*See* imported declaration entry
allocated attribute 102
anonymous union 69, 88
ARM instruction set architecture..... 108
array
  declaration of type..... 83
  descriptor for..... 221
  element ordering 83
  element type..... 83
array type entry 83
  examples 221
artificial attribute..... 34
associated attribute 102
attribute duplication..... 7
attribute ordering 7
attribute value classes..... 7
attributes..... 7
  list of 9
base address selection entry
  in location list..... 30, 31, 168
  in range list..... 38, 39, 182
base type entry..... 75
base types attribute 47
basic_block.....110, 111, 116, 119
beginning of a data member 88

beginning of an object....88, 89
big-endian encoding.... *See* endianity attribute
binary scale attribute....79
bit fields....89, 230
bit offset attribute (V3)....76, 91
bit size attrbute....75
bit size attribute....89, 98, 99, 101
bit size attribute (V3)....76, 91
bit stride attribute....83, 97, 100
block class....15, 147
block entry.... *See* try block entry, *See* lexical block entry
builtin type.... *See* base type entry
byte size attrbute....75
byte size attribute.... 89, 96, 98, 99, 101
byte size attribute (V3)....91
byte stride attribute....97, 100
C 1, 4, 35, 44, 47, 54, 55, 65, 69, 71, 75, 80, 81, 82, 84, 85, 89, 96, 97, 99, 123, 221, 222, 275, 287
C++...1, 4, 32, 33, 34, 37, 41, 44, 49, 50, 52, 57, 59, 61, 62, 64, 65, 66, 69, 70, 72, 80, 81, 82, 84, 85, 86, 87, 88, 89, 92, 93, 96, 97, 99, 100, 105, 106, 107, 123, 251, 256, 257, 260, 263, 266, 269, 271, 275, 277, 287
call column attribute....60
call file attribute....60
call frame information
  encoding....*180*
  examples....*239*
call line attribute....60
calling convention attribute....54
  encoding....*174*
case sensitivity....46
catch block entry....66
char16_t....255
char32_t....255
CIE....*See* common information entry
`CIE_id`....129, 141, 242
CIE_pointer....129, 130, 141
class template instantiation (entry)....93
class type entry....84
  as class template instantiation....93
classes of attribute value..... 7, *See also* attribute encodings
COBOL....1, 4, 99
`code_alignment_factor`....130, 132
column position of declaration....36
COMDAT....*See* section group
common (block) reference attribute....56
common block.... *See* Fortran, common block
common block entry....73
common information entry....129
compilation directory attribute....46
compilation unit....43
  for template instantiation....94
  header....143
  normal....43
  partial....43
  type....48
composite location description....28
compression....*See* DWARF compression
concrete inlined instance
  example.... 245, 248, 251
  nested....63
concrete inlined instance entry....61
concrete inlined instance root....61
concrete inlined instance tree....61
concrete out-of-line instance.... 62, 288
  example....248
  of inlined subprogram....63
condition entry....95
condition, COBOL level-88....95
const qualified type....81
constant class.... 15, 147
constant entry....69
constant expression attribute.... 60, 72
constant type entry....81
constant value attribute.... 71, 93, 96
constexpr.... 59, 61, 72
containing type attribute....100
contiguous address range....38
count attribute.... 81, 99
  default....99
D 99
data bit offset attribute.... 75, 89
data location attribute....102
data member.... *See* member entry (data)
data member location attribute.... 86, 88
debug_abbrev_offset.... 141, 144, 215
debug_info_length.... 141
debug_info_offset.... 141
debugging information entry....7
  global name for....266
  ownership relation....16
decimal scale attribute.... 78, 79
decimal sign attribute....78
DECL....191
declaration attribute.... 35, 49, 69, 85
declaration column attribute....36
declaration coordinates....36, 191, *See* also DW_AT_decl_file, DW_AT_decl_line, DW_AT_decl_column
  in concrete instance....61
declaration file attribute....36
declaration line attribute....36
default value attribute....70
default_is_stmt.... 111, 113
derived type (C++).... *See* inheritance entry
description attribute....41
descriptor, array....221
DIE.... *See* debugging information entry

digit count attribute....78, 79,
discontiguous address ranges....*See* non-contiguous address ranges
discriminant (entry) 94
discriminant attribute 94
discriminant list attribute 94
encoding *176*
discriminant value attribute....94
discriminator....111, 116, 119, 122
duplicate elimination..... *See* DWARF duplicate elimination
DW_ACCESS_private....32, 170
DW_ACCESS_protected....32, 170
DW_ACCESS_public....32, 170
DW_ADDR_far16....35
DW_ADDR_far32....35
DW_ADDR_huge16....35
DW_ADDR_near16 35
DW_ADDR_near32 35
DW_ADDR_none....34, 35, 173
DW_AT_abstract_origin . 9, 61, 62, 63, 156, 191, 192, 193, 196, 197, 198, 199, 200, 202, 203, 204, 205, 206, 207, 208, 209, 210, 247, 250, 253, 254
DW_AT_accessibility 9, 32, 87, 88, 92, 156, 170, 185, 191, 193, 195, 196, 198, 199, 200, 203, 204, 205, 206, 207, 208, 209, 210, 211, 280, 282, 283
DW_AT_address_class.... 9, 34, 55, 81, 156, 185, 195, 202, 203, 205, 206, 211
DW_AT_allocated. 9, 40, 84, 102, 103, 158, 185, 191, 192, 193, 194, 196, 197, 201, 202, 203, 204, 206, 207, 208, 209, 211, 224
DW_AT_artificial 7, 9, 34, 64, 92, 156, 185, 197, 205, 209, 236, 281
DW_AT_associated9, 40, 84, 102, 158, 185, 191, 192, 193, 194, 196, 197, 201, 202, 203, 204, 206, 207, 208, 209, 211, 223
DW_AT_base_types....9, 47, 156, 194, 201
DW_AT_binary_scale 9, 79, 158, 185, 192
DW_AT_bit_offset....9, 40, 90, 91, 155, 185, 192, 200
DW_AT_bit_offset (V3)....76, 91
DW_AT_bit_size. 9, 40, 41, 75, 83, 85, 89, 91, 98, 99, 101, 155, 185, 191, 192, 193, 196, 197, 200, 203, 204, 206, 209, 230, 231
DW_AT_bit_size (V3) 76, 91
DW_AT_bit_stride 9, 40, 41, 83, 97, 100, 156, 185, 191, 196, 206, 231
DW_AT_byte_size 9, 40, 41, 75, 76, 83, 85, 89, 90, 91, 96, 98, 99, 101, 155, 185, 191, 192, 193, 196, 197, 200, 203, 204, 206, 209, 220, 224, 254, 255, 275, 277, 278, 279, 280, 281, 282, 284
DW_AT_byte_size (V3)....76, 91
DW_AT_byte_stride.. 9, 40, 41, 83, 97, 100, 158, 185, 196, 206, 226
DW_AT_call_column....9, 60, 158, 198
DW_AT_call_file 9, 60, 158, 198
DW_AT_call_line 10, 60, 158, 198
DW_AT_calling_convention 10, 54, 156, 174, 205
DW_AT_common_reference 10, 56, 155, 193, 274, 275
DW_AT_comp_dir 10, 46, 115, 122, 155, 194, 201, 220
DW_AT_const_expr10, 60, 61, 72, 159, 185, 198, 210, 254
DW_AT_const_value. 10, 60, 61, 71, 93, 96, 103, 155, 185, 195, 196, 207, 210, 247, 254, 256, 273
DW_AT_containing_type 10, 100, 156, 185, 202
DW_AT_count....10, 40, 81, 99, 156, 185, 203, 206
DW_AT_data_bit_offset....10, 75, 76, 88, 89, 90, 91, 159, 185, 192, 200, 230, 231
DW_AT_data_location 10, 84, 102, 158, 185, 191, 192, 193, 194, 196, 197, 201, 202, 203, 204, 206, 207, 208, 209, 211, 222, 223, 224, 225, 226, 228
DW_AT_data_member_location10, 20, 86, 88, 89, 91, 156, 185, 198, 200, 224, 229, 277, 278, 279, 280, 282, 283, 284
DW_AT_data_member_location (V3)....91
DW_AT_decimal_scale 10, 78, 79, 158, 185, 192
DW_AT_decimal_sign....10, 78, 158, 169, 185, 192
DW_AT_decl_column .....10, 36, 156, 188, 191, *See* also declaration coordinates
DW_AT_decl_file.....10, 36, 157, 188, 191, 277, 278, 280, 281, *See* also declaration coordinates
DW_AT_decl_line .....10, 36, 157, 188, 191, 277, 278, 280, 281, *See* also declaration coordinates
DW_AT_declaration ..10, 35, 36, 49, 69, 85, 157, 187, 188, 191, 193, 195, 196, 200, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 236, 281
DW_AT_default_value 10, 70, 156, 185, 197
DW_AT_description.....7, 10, 41, 158, 188, 191, 192, 193, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210
DW_AT_digit_count....11, 78, 79, 158, 185, 192
DW_AT_discr....11, 94, 95, 155, 185, 210
DW_AT_discr_list....11, 94, 95, 157, 176, 185, 210
DW_AT_discr_value 11, 94, 95, 155, 185, 210
DW_AT_elemental 11, 54, 159, 205
DW_AT_encoding 11, 75, 157, 168, 186, 192, 220, 255, 275
DW_AT_endianity11, 72, 75, 159, 170, 186, 192, 195, 197, 210
DW_AT_entry_pc 11, 34, 38, 40, 49, 55, 60, 158, 198, 200, 205
DW_AT_enum_class 11, 96, 159, 186, 196, 256
DW_AT_explicit....11, 92, 159, 186, *205*
DW_AT_extension 11, 49, 158, 201, 234
DW_AT_external.... 11, 53, 69, 70, 157, 195, 205, 210, 281
DW_AT_frame_base11, 18, 29, 56, 57, 157, 195, 205, 250, 252, 253
DW_AT_friend 11, 87, 157, 187, 197
DW_AT_hi_user 159

DW_AT_high_pc11, 34, 37, 38, 44, 49, 55, 60, 65, 66, 155, 192, 194, 198, 199, 200, 201, 205, 208, 211, 220, 233, 234, 247, 250, 252, 253
DW_AT_identifier_case..... 11, 46, 157, 174, 194, 201, 273
DW_AT_import..... 11, 47, 50, 51, 155, 198, 234, 268, 274, 275
DW_AT_inline 12, 58, 59, 156, 175, 205, 246, 248, 249, 252, 254
DW_AT_is_optional..... 12, 70, 156, 186, 197
DW_AT_language. 12, 44, 48, 83, 155, 171, 194, 201, 208, 220, 270, 273, 277, 280
DW_AT_linkage_name..... 12, 37, 41, 53, 72, 73, 159, 193, 195, 205, 210
DW_AT_lo_user.....159
DW_AT_location 12, 24, 37, 60, 66, 69, 73, 155, 186, 193, 195, 197, 210, 211, 225, 229, 231, 233, 234, 236, 247, 250, 252, 253, 273, 285
DW_AT_low_pc.. 12, 34, 37, 38, 40, 44, 49, 55, 59, 60, 65, 66, 155, 192, 194, 195, 198, 199, 200, 201, 205, 208, 211, 220, 233, 234, 247, 250, 252, 253
DW_AT_lower_bound 12, 40, 99, 156, 171, 186, 206, 223, 224, 228, 229, 231, 273
DW_AT_macro_info..... 12, 45, 157, 194, 201
DW_AT_main_subprogram 3, 12, 47, 53, 159, 194, 201, 205
DW_AT_mutable 12, 88, 158, 186, 200
DW_AT_name12, 36, 37, 41, 44, 46, 49, 51, 53, 58, 62, 65, 66, 69, 73, 75, 80, 81, 82, 83, 84, 86, 87, 88, 93, 95, 96, 97, 98, 99, 100, 101, 103, 106, 107, 115, 122, 155, 184, 185, 187, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 220, 224, 225, 228, 229, 230, 231, 233, 234, 235, 236, 246, 248, 249, 252, 254, 255, 256, 257, 258, 260, 270, 271, 273, 274, 275, 277, 278, 279, 280, 281, 282, 283, 284, 285
DW_AT_namelist_item..... 12, 73, 157, 201
DW_AT_object_pointer **12**, 92, 159, 189, 205, 236
DW_AT_ordering..... 12, 83, 155, 175, 186, 191
DW_AT_picture_string 12, 78, 158, 186, 192
DW_AT_priority 12, 49, 157, 200
DW_AT_producer 12, 46, 156, 194, 201, 220
DW_AT_prototyped..... 12, 54, 97, 156, 186, 205, 206
DW_AT_pure 12, 55, 159, 205
DW_AT_ranges..12, 34, 37, 38, 44, 49, 55, 60, 65, 66, 158, 192, 194, 198, 199, 200, 201, 205, 208, 211
DW_AT_recursive..... 13, 54, 55, 159, 205
DW_AT_return_addr..... 13, 56, 60, 156, 195, 198, 205
DW_AT_segment.... 13, 34, 55, 60, 70, 157, 186, 192, 193, 194, 195, 197, 198, 199, 200, 201, 205, 208, 210, 211
DW_AT_sibling .. 13, 16, 36, 155, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211
DW_AT_signature..... 13, 85, 159, 193, 196, 204, 207, 209
DW_AT_small..... 13, 79, 158, 186, 192
DW_AT_specification ... 13, 36, 50, 59, 70, 85, 92, 93, 157, 187, 191, 193, 196, 200, 204, 205, 209, 210, 234
DW_AT_start_scope.. 13, 37, 38, 60, 71, 75, 156, 191, 193, 195, 196, 197, 198, 199, 201, 203, 204, 205, 206, 207, 208, 209, 210
DW_AT_static_link 13, 56, 57, 157, 195, 205, 247, 250, 252
DW_AT_stmt_list 13, 45, 155, 194, 201, 220
DW_AT_string_length.....13, 98, 155, 186, 204
DW_AT_threads_scaled 13, 99, 159, 186, 206
DW_AT_trampoline.....13, 64, 158, 198, 205
DW_AT_type. 13, 32, 55, 57, 58, 66, 70, 72, 81, 82, 83, 86, 88, 93, 94, 95, 96, 97, 98, 99, 100, 101, 103, 157, 187, 191, 194, 195, 196, 197, 198, 200, 201, 202, 203, 205, 206, 207, 208, 210, 211, 220, 223, 224, 225, 228, 229, 230, 231, 233, 234, 235, 236, 246, 249, 252, 254, 255, 256, 257, 258, 260, 270, 271, 273, 274, 277, 278, 279, 280, 281, 282, 283, 284, 285
DW_AT_upper_bound. 13, 40, 99, 156, 186, 206, 223, 224, 228, 229, 231, 254, 273
DW_AT_use_location.....13, 100, 101, 157, 186, 202
DW_AT_use_UTF8 .....13, 47, 150, 158, 186, 194, 201
DW_AT_variable_parameter 13, 70, 157, 186, 197
DW_AT_virtuality .. 13, 33, 87, 92, 157, 171, 186, 198, 205
DW_AT_visibility..... 14, 33, 155, 171, 186, 191, 193, 195, 196, 197, 200, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211
DW_AT_vtable_elem_location.....14, 92, 157, 186, 205
DW_ATE_address 77, 168
DW_ATE_boolean..... 77, 168
DW_ATE_complex_float 77, 168
DW_ATE_decimal_float..... 77, 169
DW_ATE_edited..... 77, 78, 168
DW_ATE_float..... 77, 168
DW_ATE_hi_user..... 169
DW_ATE_imaginary_float..... 77, 168
DW_ATE_lo_user..... 169
DW_ATE_numeric_string 77, 78, 79, 168
DW_ATE_packed_decimal..... 77, 78, 79, 168
DW_ATE_signed..... 75, 77, 168, 275
DW_ATE_signed_char..... 77, 168
DW_ATE_signed_fixed..... 77, 78, 169
DW_ATE_unsigned..... 77, 168, 220
DW_ATE_unsigned_char 77, 168, 220
DW_ATE_unsigned_fixed..... 77, 78, 169
DW_ATE_UTF..... 77, 78, 169, 255
DW_CC_hi_user..... 174
DW_CC_lo_user..... 174
DW_CC_nocall..... 54, 174
DW_CC_normal 54, 174
DW_CC_program 54, 174
DW_CFA_advance_loc 132, 136, 137, 181, 243
DW_CFA_advance_loc1..... 132, 181
DW_CFA_advance_loc2..... 132, 181

DW_CFA_advance_loc4 ..... 132, 181
DW_CFA_def_cfa 131, 132, 133, 181, 242, 243
DW_CFA_def_cfa_expression 131, 133, 181
DW_CFA_def_cfa_offset 133, 181, 243
DW_CFA_def_cfa_offset_sf 133, 182
DW_CFA_def_cfa_register 133, 181, 243
DW_CFA_def_cfa_sf 133, 182
DW_CFA_expression 131, 135, 181
DW_CFA_hi_user 182
DW_CFA_lo_user 182
DW_CFA_nop 130, 131, 136, 181, 242, 243
DW_CFA_offset 134, 181, 243
DW_CFA_offset_extended 134, 181, 182
DW_CFA_offset_extended_sf 134, 182
DW_CFA_register 135, 181, 242
DW_CFA_remember_state 136, 181
DW_CFA_restore 136, 181, 243
DW_CFA_restore_extended 136, 181
DW_CFA_restore_state 136, 181
DW_CFA_same_value 134, 181, 242
DW_CFA_set_loc 132, 136, 137, 181
DW_CFA_undefined 134, 137, 181, 242
DW_CFA_val_expression 131, 135, 182
DW_CFA_val_offset 134, 135, 182
DW_CFA_val_offset_sf 135, 182
DW_CHILDREN_no 146, 154, 220
DW_CHILDREN_yes 146, 154, 220
DW_DS_leading_overpunch 80, 169
DW_DS_leading_separate 80, 169
DW_DS_trailing_overpunch 80, 169
DW_DS_trailing_separate 80, 169
DW_DS_unsigned 80, 169
DW_DSC_label 95, 176
DW_DSC_range 95, 176
DW_END_big 72, 170
DW_END_default 72, 170
DW_END_hi_user 170
DW_END_little 72, 170
DW_END_lo_user 170
DW_FORM_addr 147, 160, 168, 182, 220
DW_FORM_block 135, 147, 160, 187
DW_FORM_block1 ..... 147, 160
DW_FORM_block2 ..... 147, 160
DW_FORM_block4 ..... 147, 160
DW_FORM_data1 ..... 147, 160, 220
DW_FORM_data2 ..... 147, 160
DW_FORM_data4 ..... 3, 146, 147, 148, 160, 215
DW_FORM_data8 ..... 3, 146, 147, 148, 160
DW_FORM_exprloc 133, 148, 161
DW_FORM_flag 148, 160, 187
DW_FORM_flag_present 148, 161
DW_FORM_indirect 146, 161, 220
DW_FORM_ref_addr 24, 142, 149, 150, 160, 265, 267, 268, 288
DW_FORM_ref_sig8 ..... 145, 161
DW_FORM_ref_udata 149, 161, 267
DW_FORM_ref1 ..... 149, 160, 267
DW_FORM_ref2 ..... 24, 149, 160, 267
DW_FORM_ref4 ..... 24, 149, 160, 220, 267
DW_FORM_ref8 ..... 149, 160, 267
DW_FORM_sdata 147, 160, 187
DW_FORM_sec_offset 3, 142, 146, 148, 149, 161, 220
DW_FORM_string 150, 160, 187, 220
DW_FORM_strp 142, 150, 160, 215
DW_FORM_udata 147, 160
DW_ID_case_insensitive 46, 174, 273
DW_ID_case_sensitive 46, 174
DW_ID_down_case 46, 174
DW_ID_up_case 46, 174
DW_INL_declared_inlined 59, 175, 246, 249, 252
DW_INL_declared_not_inlined 59, 175
DW_INL_inlined 59, 175
DW_INL_not_inlined 59, 175
DW_LANG_Ada83 ..... 44, 172
DW_LANG_Ada95 ..... 44, 172
DW_LANG_C 44, 172, 220
DW_LANG_C_plus_plus 44, 172, 270, 277, 280
DW_LANG_C89 ..... 44, 172, 220
DW_LANG_C99 ..... 44, 172
DW_LANG_Cobol74 ..... 44, 172
DW_LANG_Cobol85 ..... 44, 172
DW_LANG_Fortran77 ..... 44, 172
DW_LANG_Fortran90 ..... 44, 172, 273
DW_LANG_Fortran95 ..... 45, 172
DW_LANG_hi_user 173
DW_LANG_Java 45, 172
DW_LANG_lo_user 173
DW_LANG_Modula2 ..... 45, 172
DW_LANG_ObjC 45, 173
DW_LANG_ObjC_plus_plus 45, 173
DW_LANG_Pascal83 ..... 45, 172
DW_LANG_PLI 45, 172
DW_LANG_Python 45, 173
DW_LANG_UPC 173
DW_LNE_define_file 115, 122, 179
DW_LNE_end_sequence 121, 179, 238
DW_LNE_hi_user 179
DW_LNE_lo_user 179
DW_LNE_set_address 121, 179
DW_LNE_set_discriminator 122, 179
DW_LNS_advance_line 119, 178
DW_LNS_advance_pc 119, 120, 178, 238
DW_LNS_const_add_pc 120, 178
DW_LNS_copy 119, 178
DW_LNS_fixed_advance_pc 111, 120, 178, 238
DW_LNS_hi_user omission 139
DW_LNS_lo_user omission 139
DW_LNS_negate_stmt 113, 119, 178

DW_LNS_set_basic_block....119, 178
DW_LNS_set_column....119, 178
DW_LNS_set_epilogue_begin....121, 179
DW_LNS_set_file....119, 178
DW_LNS_set_isa....121, 179
DW_LNS_set_prologue_end....120, 178
DW_MACINFO_define....123, 124, 125, 180
DW_MACINFO_end_file....123, 124, 180
DW_MACINFO_start_file....123, 124, 125, 180
DW_MACINFO_undef....123, 125, 180
DW_MACINFO_vendor_ext....123, 124, 180
DW_OP_abs....21, 164
DW_OP_addr....17, 29, 163
DW_OP_and....21, 164, 223, 224
DW_OP_bit_piece....29, 167
DW_OP_bra....23, 165
DW_OP_breg0....18, 56, 166
DW_OP_breg1....18, 30, 166
DW_OP_breg11....29
DW_OP_breg2....30
DW_OP_breg3....30
DW_OP_breg31....18, 166
DW_OP_breg4....30
DW_OP_bregx....18, 27, 29, 166
DW_OP_call_frame_cfa....21, 131, 167
DW_OP_call_ref....24, 37, 131, 167, 188, 215
DW_OP_call2....24, 37, 131, 167
DW_OP_call4....24, 37, 131, 167
DW_OP_const1s....18, 164
DW_OP_const1u....17, 164
DW_OP_const2s....18, 164
DW_OP_const2u....17, 164
DW_OP_const4s....18, 164
DW_OP_const4u....17, 164
DW_OP_const8s....18, 164
DW_OP_const8u....17, 164
DW_OP_consts....18, 164
DW_OP_constu....18, 164
DW_OP_deref....19, 29, 163, 223, 224
DW_OP_deref_size....19, 166
DW_OP_div....21, 164
DW_OP_drop....18, 25, 164
DW_OP_dup....18, 25, 164
DW_OP_eq....23, 165
DW_OP_fbreg....18, 29, 30, 166
DW_OP_form_tls_address....20, 167
DW_OP_ge....23, 165
DW_OP_gt....23, 165
DW_OP_hi_user....167
DW_OP_implicit_value....28, 167
DW_OP_le....23, 165
DW_OP_lit0....17, 166
DW_OP_lit1....17, 30, 166, 223
DW_OP_lit2....17, 224
DW_OP_lit31....17, 166
DW_OP_lit*n*....17, 22, 223, 224, 229
DW_OP_lo_user....167
DW_OP_lt....23, 165
DW_OP_minus....21, 165
DW_OP_mod....21, 165
DW_OP_mul....21, 165
DW_OP_ne....23, 165
DW_OP_neg....22, 165
DW_OP_nop....24, 166
DW_OP_not....22, 165
DW_OP_or....22, 165
DW_OP_over....19, 25, 164
DW_OP_pick....19, 25, 164
DW_OP_piece....28, 30, 166
DW_OP_plus....22, 30, 165, 223, 224, 229
DW_OP_plus_uconst....22, 30, 165
DW_OP_push_object_address...20, 88, 102, 131, 167, 223, 224, 225, 226
DW_OP_reg0....27, 30, 56, 166
DW_OP_reg1....27, 166
DW_OP_reg10....30
DW_OP_reg3....29, 30
DW_OP_reg31....27, 166
DW_OP_regx....27, 29, 166
DW_OP_rot....19, 25, 164
DW_OP_shl....22, 165
DW_OP_shr....22, 165
DW_OP_shra....22, 165
DW_OP_skip....23, 165
DW_OP_stack_value....28, 30, 167
DW_OP_swap....19, 25, 164
DW_OP_xderef....19, 164
DW_OP_xderef_size....20, 166
DW_OP_xor....22, 165
DW_ORD_col_major....83, 175
DW_ORD_row_major....83, 175
DW_TAG_access_declaration....8, 87, 152, 191
DW_TAG_array_type..8, 83, 151, 191, 223, 224, 228, 229, 231, 254, 273
DW_TAG_base_type.....8, 75, 82, 152, 192, 220, 230, 233, 235, 255, 270, 275, 278, 279, 281, 282
DW_TAG_catch_block....8, 66, 152, 192
DW_TAG_class_type.....8, 84, 93, 151, 193, 235, 280, 282
DW_TAG_common_block....8, 41, 73, 152, 193, 273
DW_TAG_common_inclusion.....8, 56, 152, 193, 274, 275
DW_TAG_compile_unit.....8, 43, 143, 151, 194, 220, 267, 268, 270, 271, 274, 275, 285
DW_TAG_condition....8, 95, 153, 194
DW_TAG_const_type....8, 81, 82, 152, 194, 235, 254
DW_TAG_constant....8, 41, 69, 79, 95, 152, 195, 273
DW_TAG_dwarf_procedure....8, 37, 153, 195
DW_TAG_entry_point....8, 41, 53, 151, 195
DW_TAG_enumeration_type....8, 84, 96, 151, 196, 256

DW_TAG_enumerator 8, 96, 152, 196, 256
DW_TAG_file_type 8, 101, 152, 197
DW_TAG_formal_parameter 8, 67, 69, 95, 97, 151, 197, 236, 246, 247, 249, 250, 252, 253, 254, 258, 281
DW_TAG_friend 8, 87, 152, 187, 197
DW_TAG_hi_user 139, 154
DW_TAG_imported_declaration 8, 50, 151, 198, 234
DW_TAG_imported_module 8, 51, 153, 198, 234
DW_TAG_imported_unit 8, 47, 153, 198, 268, 274, 275, 287
DW_TAG_inheritance 8, 86, 152, 198
DW_TAG_inlined_subroutine 8, 53, 60, 61, 63, 64, 152, 198, 246, 247, 250, 253, 254
DW_TAG_interface_type 8, 86, 153, 199
DW_TAG_label 8, 65, 151, 199
DW_TAG_lexical_block 8, 65, 151, 199
DW_TAG_lo_user 139, 154
DW_TAG_member . 8, 70, 88, 95, 151, 200, 224, 228, 229, 230, 231, 257, 258, 260, 270, 277, 278, 279, 280, 282, 283, 284
DW_TAG_module 8, 49, 152, 200
DW_TAG_namelist 8, 73, 153, 200
DW_TAG_namelist_item 8, 73, 153, 201
DW_TAG_namespace . 8, 49, 153, 201, 233, 234, 277, 279, 280, 281, 282, 283
DW_TAG_packed_type 8, 81, 153, 201
DW_TAG_partial_unit 8, 43, 143, 153, 201, 268, 271, 273, 275
DW_TAG_pointer_type 8, 81, 82, 151, 187, 202, 220, 235, 281, 283
DW_TAG_ptr_to_member_type 8, 100, 152, 187, 202
DW_TAG_reference_type 8, 81, 151, 187, 202, 271
DW_TAG_restrict_type 8, 81, 82, 153, 202
DW_TAG_rvalue_reference_type 8, 81, 154, 187, 203
DW_TAG_set_type 8, 98, 152, 203
DW_TAG_shared_type 8, 81, 154, 203
DW_TAG_string_type 8, 98, 151, 204
DW_TAG_structure_type 8, 84, 93, 151, 204, 224, 229, 230, 231, 257, 258, 260, 261, 270, 277, 279, 281, 284
DW_TAG_subprogram 8, 41, 53, 58, 59, 61, 63, 64, 92, 153, 187, 205, 233, 234, 236, 246, 247, 249, 250, 252, 253, 254, 258, 271, 274, 275, 281, 284, 285
DW_TAG_subrange_type 8, 84, 95, 99, 152, 171, 206, 223, 224, 228, 229, 231, 254, 273
DW_TAG_subroutine_type 8, 97, 151, 206
DW_TAG_template_alias 8, 103, 154, 207, 260, 261
DW_TAG_template_type_parameter 8, 58, 93, 103, 153, 207, 257, 258, 260, 261
DW_TAG_template_value_parameter . 8, 93, 103, 153, 207
DW_TAG_thrown_type 8, 57, 153, 207
DW_TAG_try_block 8, 66, 153, 208
DW_TAG_type_unit 8, 48, 154, 208, 277, 280
DW_TAG_typedef 8, 82, 151, 208, 220
DW_TAG_union_type 8, 84, 93, 152, 209
DW_TAG_unspecified_parameters .. 8, 56, 67, 97, 152, 209
DW_TAG_unspecified_type 8, 80, 153, 209, 235
DW_TAG_variable 8, 41, 61, 69, 82, 95, 153, 210, 225, 228, 229, 231, 233, 234, 246, 247, 249, 250, 252, 253, 254, 255, 256, 257, 258, 260, 261, 271, 273, 274, 285
DW_TAG_variant 8, 94, 152, 153, 210
DW_TAG_variant_part 8, 94, 153, 210
DW_TAG_volatile_type 8, 81, 82, 153, 211
DW_TAG_with_stmt 8, 66, 152, 211
DW_VIRTUALITY_none 33, 171
DW_VIRTUALITY_pure_virtual 33, 171
DW_VIRTUALITY_virtual 33, 171
DW_VIS_exported 33, 171
DW_VIS_local 33, 171
DW_VIS_qualified 33, 171
DWARF compression 263
DWARF duplicate elimination 263
  C example 275
  C++ example 269
  examples 269
  Fortran example 272
DWARF expression ..... 17, *See also* location description
  arithmetic operations 21
  control flow operations 23
  examples 25
  literal encodings 17
  logical operations 21
  operator encodings 163
  special operations 24
  stack operations 17
DWARF procedure 37
DWARF procedure entry 37
DWARF section names, list of 183
DWARF Version 2 ..... 4, 5, 114, 140, 289
DWARF Version 3 .. 1, 2, 3, 4, 38, 55, 76, 91, 114, 117, 289
elemental attribute 54
empty location description 28
encoding attribute 75
  encoding 168
end of list entry
  in location list 31, 168
  in range list 38, 182
end_sequence 110, 111, 121
endianity attribute 72, 75
entity 7
entry PC attribute 34
  and abstract instance 60
  for inlined subprogram 60
  for module initialization 49
  for subroutine 55
entry point entry 53
enum class ..... *See* type-safe enumeration
enumeration literal ..... *See* enumerator entry
enumeration type entry 96

as array dimension 84, 97
enumerator entry 96
epilogue 116, 121, 126, 127, 136, 179, 240
epilogue_begin 110, 111, 121
epilogue_end 119
error value 140
exception, thrown ..... *See* thrown type entry
explicit attribute 92
exprloc class 15, 26, 148
extended type (Java) ..... *See* inheritance entry
extensibility ..... *See* vendor extensibility
extension attribute 49
external attribute 53, 69
FDE ..... *See* frame description entry
file containing declaration 36
file type entry 101
file_names 115
flag class 15, 148
formal parameter 55
formal parameter entry 69, 97
in catch block 67
with default value 70
formal type parameter ..... *See* template type parameter entry
Fortran 1, 4, 44, 47, 52, 53, 54, 73, 98, 99, 102, 221, 268, 272
common block 56, 73
main program 54
module (Fortran 90) 49
use statement 51, 52
frame base attribute 56
frame description entry 130
friend attribute 87
friend entry 87
function entry ..... *See* subroutine entry
fundamental type ..... *See* base type entry
global namespace ..... See namespace (C++), global
header_length 141
hidden indirection ..... *See* data location attribute
high PC attribute 34, 37, 38, 44, 49, 55, 60, 65, 66
and abstract instance 60
identifier case attribute 46
encoding *174*
identifier names 36
implementing type (Java) ..... *See* inheritance entry
implicit location description 27
import attribute 47, 50, 51
imported declaration entry 50
imported module entry 51
imported unit entry 43, 47
include_directories 114, 115, 122
incomplete class/structure/union 85
incomplete declaration *35*
incomplete type 85
inheritance entry 86
initial length 143, 144
initial length field 106, 107, 112, 129, 130, 176, 177
encoding 140
inline attribute 58, 59
encoding *175*
inlined subprogram call
examples 244
inlined subprogram entry 53, 60
in concrete instance 61
interface type entry 86
is optional attribute 70
is_stmt 110, 111, 113, 119
isa 17, 111, 121
Java 4, 45, 84, 86, 99
label entry 65
language attribute 44, 83
language name encoding *171*
LEB128
examples 162
signed, decoding of *218*
signed, encoding as 161, *217*
unsigned, decoding of *218*
unsigned, encoding as 162, *217*
level-88 condition, COBOL 95
lexical block entry 65
line number information ..... *See also* statement list attribute
line number of declaration 36
line number opcodes
extended opcode encoding 179
standard opcode encoding 178
line_base 113, 116, 117, 118, 237
line_range 113, 116, 117, 118, 237
lineptr 151
lineptr class 15, 148
linkage name attribute 41
Little Endian Base 128 ..... *See* LEB128
little-endian encoding ..... *See* endian attribute
location attribute 37, 66, 69, 73
and abstract instance 60
location description 30
location description ..... 26, *See also* DWARF expression
composite 28
empty 28
implicit 27
memory 27
simple 26
single 26
location description
use in location list 31
location description 88
location list 26, 30, 56, 148, 167, 215
base address selection entry 31
end of list entry 31
entry 30

loclistptr 151
loclistptr class 15, 26, 148
lookup
  by address 107
  by name 106
low PC attribute 34, 37, 38, 44, 49, 55, 60, 65, 66
  and abstract instance 59
lower bound attribute 99
  default 99, 171
macinfo types 123
  encoding *180*
macptr 151
macptr class 15, 149
macro formal parameter list 124
macro information 123
macro information attribute 45
main subprogram attribute 47, 53
mangled names 41
maximum_operations_per_instruction 112, 113, 117, 118
MD5 hash 184, 188, 189, 280, 284
member entry (data) 88
  as discriminant 94
member function entry 92
memory location description 27
minimum_instruction_length .. 112, 113, 117, 118, 120, 237
MIPS instruction set architecture 108
Modula-2 ..... 33, 45, 49, 66, 99
  definition module 49
module entry 49
mutable attribute 88
name attribute .36, 41, 44, 46, 49, 51, 53, 58, 62, 65, 66, 69, 73, 75, 80, 81, 82, 83, 84, 86, 87, 88, 93, 96, 97, 98, 99, 100, 101, 106
namelist entry 73
namelist item attribute 73
namelist item entry 73
names
  identifier 36
  mangled 41
namespace (C++) 49
  alias 51
  example 232
  global 50
  std 50
  unnamed 50
  using declaration 51
  using directive 52
namespace declaration entry 49
namespace extension entry 49
nested abstract instance 63
nested concrete inline instance 63
non-contiguous address ranges 38
non-defining declaration 35
normal compilation unit 43
object pointer attribute **92**
Objective C 45, 92, 99
Objective C++ 45
Objective C++, ... 99
op_index 110, 111, 112, 113, 116, 117, 119, 120, 121
opcode_base 114, 116, 117, 237
operation advance 117, 119
operation pointer 110, 113, 116, 117
optional parameter 70
ordering attribute 83
  encoding *175*
out-of-line instance ..... *See* concrete out-of-line instance
packed type entry 81
parameter ..... *See* macro formal parameter list, *See* `this` parameter, *See* variable parameter attribute, *See* optional parameter attribute, *See* unspecified parameters entry, *See* template value parameter entry, *See* template type parameter entry, *See* formal parameter entry
partial compilation unit 43
Pascal 45, 66, 81, 84, 98, 99, 101
PL/I 99
pointer to member type entry 100
pointer type entry 81
priority attribute 49
producer attribute 46
PROGRAM statement 47, 53
prologue 4, 116, 120, 121, 126, 127, 178, 240
prologue_end 110, 111, 119, 120
prototyped attribute 54, 97
pure attribute 55
range list 38, 182, 215
rangelistptr 151
rangelistptr class 15, 149
ranges attribute 34, 38, 44, 49, 55, 60, 65, 66
  and abstract instance 60
recursive attribute 55
reference class 15, 149
reference type entry 81
  lvalue ..... *See* reference type entry
  rvalue ..... *See* rvalue reference type entry
renamed declaration ..... *See* imported declaration entry
restrict qualified type 81
restricted type entry 81
return address attribute 56
  and abstract instance 60
return type of subroutine 55
rvalue reference type entry 81
sbyte 105, 113, 184
section group 264, 267, 269, 270, 273
  name 266
section length
  in .debug_aranges header 107
  in .debug_pubnames header 106, 177
  in .debug_pubtypes header 106, 177

use in headers 141
section offset
alignment of 183
in .debug_info header 144
in .debug_pubnames header 106, 176, 177
in .debug_pubnames offset/name pair 106
in .debug_pubtypes header 106
in .debug_pubtypes name/offset pair 106
in class lineptr value 148
in class loclistptr value 148
in class macptr value 149
in class rangelistptr value 149
in class reference value 149
in class string value 150
in FDE header 130
in macro information attribute 45
in statement list attribute 45
use in headers 141
segment attribute 34, 55
and abstract instance 60
and data segment 70
`segment_size` 107, 129, 131, 132, 177, 178
segmented addressing *See* address space, *See* address space
self pointer attribute ..... *See* object pointer attribute
set type entry 98
shared qualified type 81
shared qualified type entry 81
sibling attribute 16
simple location description 26
single location description 26
size of an address ... 16, 17, 19, 20, 30, 31, 39, 98, 107, 144, 177, 178
small attribute 79
specification attribute 36, 70, 85, 92
standard_opcode_lengths 114
start scope attribute 71, 75
and abstract instance 60
statement list attribute 45
static link attribute 57
stride attribute *See* bit stride attribute or byte stride attribute
string class 15, 150
string length attribute 98
string type entry 98
structure type entry 84
subprogram entry 53
as member function 92
use for template instantiation 58
use in inlined subprogram 58
subrange type entry 99
as array dimension 84
subroutine type entry 97
tag 7
tag names ..... *See also debugging information entry*
list of 7
Template alias entry 103
template example 257
template instantiation 58
and special compilation unit 94
template type parameter entry 58, 93
template value parameter entry 93
`this` parameter 34, 64
this pointer attribute ..... *See* object pointer attribute
thread-local storage 20
threads scaled attribute 99
thrown exception ..... *See* thrown type entry
thrown type entry 57
trampoline (subroutine) entry 64
trampoline attribute 64
try block entry 66
type attribute .. 32, 55, 57, 58, 66, 70, 81, 82, 83, 86, 88, 93, 94, 97, 98, 100, 101
type modifier entry . *See* shared type entry, *See* volatile type entry, *See* reference type entry, *See* restricted type entry, *See* pointer type entry, *See* packed type entry, *See* constant type entry
type safe enumeration types 96
type signature 13, 150, 184, 188, 189, 276, 282, 288
computation grammar 285
example computation 277
type unit 43, 48, 85, 144, 145, 150, 184, 188, 276, 281, 288
type_offset 141, 145
type_signature 145
typedef entry 82
type-safe enumeration 256
ubyte 105, 107, 111, 112, 113, 114, 116, 129, 130, 131, 132, 144, 177, 184
uhalf 105, 106, 107, 112, 120, 132, 143, 144, 176, 177, 184
unallocated variable 69
Unicode character encodings 255
union type entry 84
unit ..... *See* compilation unit
unit_length 106, 107, 112, 143, 144, 176, 177
unnamed namespace ..... See namespace (C++), unnamed
unspecified parameters attribute 56
unspecified parameters entry 97
in catch block 67
unspecified type entry 80
UPC 81, 99
uplevel address ..... *See* static link attribute
upper bound attribute 99
default 99
use location attribute 100
use statement ... *See* Fortran, use statement, *See* Fortran, use statement
use UTF-8 attribute ..... 47, See also UTF-8
using declaration .. See namespace (C++), using declaration
using directive ..... See namespace (C++), using directive
UTF-8 ..... 4, 13, 47, 129, 150

uword 105, 132, 184
variable entry 69
  examples *221*
  in concrete instance 61
variable length data ..... 161, *See also* LEB128
variable parameter attribute 70
variant entry 94
variant part entry 94
vendor extensibility 2, 114, 139
vendor extension ..... 251, *See also* vendor extensibility
  for macro information 124
vendor id 139
vendor specifc extensions ..... *See* vendor extensibility
version number 289
  address lookup table 107, 177
  call frame information 129, 180, 242
  debug information 143, 144, 220
  line number information 112, 178, 237
  name lookup table 106, 176
virtuality attribute 33, 87, 92
  encoding *171*
visibility attribute 33
  encoding 171
void type ..... *See* unspecified type entry
volatile qualified type 81
volatile type entry 81
vtable element location attribute 92
with statement entry 66