

DWARF Debugging Information Format

Version 3



[DWARF Debugging Information Format Workgroup](http://dwarf.freestandards.org)

<http://dwarf.freestandards.org>



[Free Standards Group](http://freestandards.org)

<http://freestandards.org>

December 20, 2005

DWARF Debugging Information Format, Version 3

Copyright © 2005 Free Standards Group

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

This document is based 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)

Copyright © 1992, 1993 UNIX International, Inc.

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

Trademarks:

Intel386 is a trademark of Intel Corporation.

Java is a trademark of Sun Microsystems, Inc.

All other trademarks found herein are property of their respective owners.

Table of Contents

DWARF DEBUGGING INFORMATION FORMAT VERSION 3	I
1 INTRODUCTION	1
1.1 PURPOSE AND SCOPE.....	1
1.2 OVERVIEW	1
1.3 VENDOR EXTENSIBILITY	2
1.4 CHANGES FROM VERSION 1 TO VERSION 2.....	2
1.5 CHANGES FROM VERSION 2 TO VERSION 3.....	3
1.5.1 <i>Upward Compatibility</i>	4
2 GENERAL DESCRIPTION	5
2.1 THE DEBUGGING INFORMATION ENTRY	5
2.2 ATTRIBUTE TYPES.....	5
2.3 RELATIONSHIP OF DEBUGGING INFORMATION ENTRIES	13
2.4 TARGET ADDRESSES	13
2.5 DWARF EXPRESSIONS	14
2.5.1 <i>General Operations</i>	14
2.5.2 <i>Example Stack Operations</i>	22
2.6 LOCATION DESCRIPTIONS	22
2.6.1 <i>Register Name Operators</i>	23
2.6.2 <i>Simple Location Expressions</i>	23
2.6.3 <i>Composition Operators</i>	24
2.6.4 <i>Location Expressions</i>	25
2.6.5 <i>Example Location Expressions</i>	25
2.6.6 <i>Location Lists</i>	26
2.7 TYPES OF DECLARATIONS	27
2.8 ACCESSIBILITY OF DECLARATIONS	28
2.9 VISIBILITY OF DECLARATIONS	28
2.10 VIRTUALITY OF DECLARATIONS.....	29
2.11 ARTIFICIAL ENTRIES	29
2.12 SEGMENTED ADDRESSES.....	29
2.13 NON-DEFINING DECLARATIONS AND COMPLETIONS.....	30
2.13.1 <i>Non-Defining Declarations</i>	31
2.13.2 <i>Declarations Completing Non-Defining Declarations</i>	31
2.14 DECLARATION COORDINATES	31
2.15 IDENTIFIER NAMES.....	32
2.16 DATA LOCATIONS	32
2.17 CODE ADDRESSES AND RANGES	32
2.17.1 <i>Single Address</i>	32
2.17.2 <i>Contiguous Address Range</i>	33
2.17.3 <i>Non-Contiguous Address Ranges</i>	33
2.18 ENTRY ADDRESS	34
2.19 STATIC AND DYNAMIC PROPERTIES OF TYPES	35
2.20 ENTITY DESCRIPTIONS	35

DWARF Debugging Information Format, Version 3

3	PROGRAM SCOPE ENTRIES.....	37
3.1	COMPILATION AND IMPORTING ENTRIES.....	37
3.1.1	<i>Normal and Partial Compilation Unit Entries.....</i>	37
3.1.2	<i>Imported Unit Entries.....</i>	41
3.2	MODULE, NAMESPACE AND IMPORTING ENTRIES.....	41
3.2.1	<i>Module Entries.....</i>	41
3.2.2	<i>Namespace Entries.....</i>	42
3.2.3	<i>Imported (or Renamed) Declaration Entries.....</i>	43
3.2.4	<i>Imported Module Entries.....</i>	44
3.3	SUBROUTINE AND ENTRY POINT ENTRIES.....	45
3.3.1	<i>General Subroutine and Entry Point Information.....</i>	45
3.3.2	<i>Subroutine and Entry Point Return Types.....</i>	47
3.3.3	<i>Subroutine and Entry Point Locations.....</i>	47
3.3.4	<i>Declarations Owned by Subroutines and Entry Points.....</i>	48
3.3.5	<i>Low-Level Information.....</i>	48
3.3.6	<i>Types Thrown by Exceptions.....</i>	49
3.3.7	<i>Function Template Instantiations.....</i>	50
3.3.8	<i>Inline Subroutines.....</i>	50
3.3.9	<i>Trampolines.....</i>	56
3.4	LEXICAL BLOCK ENTRIES.....	57
3.5	LABEL ENTRIES.....	57
3.6	WITH STATEMENT ENTRIES.....	58
3.7	TRY AND CATCH BLOCK ENTRIES.....	58
4	DATA OBJECT AND OBJECT LIST ENTRIES.....	59
4.1	DATA OBJECT ENTRIES.....	59
4.2	COMMON BLOCK ENTRIES.....	62
4.3	NAMELIST ENTRIES.....	62
5	TYPE ENTRIES.....	63
5.1	BASE TYPE ENTRIES.....	63
5.2	UNSPECIFIED TYPE ENTRIES.....	67
5.3	TYPE MODIFIER ENTRIES.....	67
5.4	TYPEDDEF ENTRIES.....	69
5.5	ARRAY TYPE ENTRIES.....	69
5.6	STRUCTURE, UNION, CLASS AND INTERFACE TYPE ENTRIES.....	70
5.6.1	<i>Structure, Union and Class Type Entries.....</i>	71
5.6.2	<i>Interface Type Entries.....</i>	72
5.6.3	<i>Derived or Extended Classes and Interfaces.....</i>	73
5.6.4	<i>Access Declarations.....</i>	73
5.6.5	<i>Friends.....</i>	74
5.6.6	<i>Data Member Entries.....</i>	74
5.6.7	<i>Member Function Entries.....</i>	77
5.6.8	<i>Class Template Instantiations.....</i>	78
5.6.9	<i>Variant Entries.....</i>	79
5.7	CONDITION ENTRIES.....	80
5.8	ENUMERATION TYPE ENTRIES.....	80
5.9	SUBROUTINE TYPE ENTRIES.....	81

5.10	STRING TYPE ENTRIES	82
5.11	SET ENTRIES	83
5.12	SUBRANGE TYPE ENTRIES	83
5.13	POINTER TO MEMBER TYPE ENTRIES	84
5.14	FILE TYPE ENTRIES	86
5.15	DYNAMIC TYPE PROPERTIES	86
5.15.1	<i>Data Location</i>	86
5.15.2	<i>Allocation and Association Status</i>	86
5.16	DWARF PROCEDURES	87
6	OTHER DEBUGGING INFORMATION	89
6.1	ACCELERATED ACCESS	89
6.1.1	<i>Lookup by Name</i>	90
6.1.2	<i>Lookup by Address</i>	91
6.2	LINE NUMBER INFORMATION	92
6.2.1	<i>Definitions</i>	93
6.2.2	<i>State Machine Registers</i>	93
6.2.3	<i>Line Number Program Instructions</i>	95
6.2.4	<i>The Line Number Program Header</i>	95
6.2.5	<i>The Line Number Program</i>	98
6.3	MACRO INFORMATION	104
6.3.1	<i>Macinfo Types</i>	104
6.3.2	<i>Base Source Entries</i>	106
6.3.3	<i>Macinfo Entries for Command Line Options</i>	106
6.3.4	<i>General Rules and Restrictions</i>	106
6.4	CALL FRAME INFORMATION	107
6.4.1	<i>Structure of Call Frame Information</i>	108
6.4.2	<i>Call Frame Instructions</i>	112
6.4.3	<i>Call Frame Instruction Usage</i>	117
6.4.4	<i>Call Frame Calling Address</i>	118
7	DATA REPRESENTATION	119
7.1	VENDOR EXTENSIBILITY	119
7.2	RESERVED VALUES	120
7.2.1	<i>Error Values</i>	120
7.2.2	<i>Initial Length Values</i>	120
7.3	EXECUTABLE OBJECTS AND SHARED OBJECTS	120
7.4	32-BIT AND 64-BIT DWARF FORMATS	120
7.5	FORMAT OF DEBUGGING INFORMATION	123
7.5.1	<i>Compilation Unit Header</i>	123
7.5.2	<i>Debugging Information Entry</i>	124
7.5.3	<i>Abbreviations Tables</i>	124
7.5.4	<i>Attribute Encodings</i>	125
7.6	VARIABLE LENGTH DATA	139
7.7	DWARF EXPRESSIONS AND LOCATION DESCRIPTIONS	141
7.7.1	<i>DWARF Expressions</i>	141
7.7.2	<i>Location Expressions</i>	145
7.7.3	<i>Location Lists</i>	145
7.8	BASE TYPE ENCODINGS	146

DWARF Debugging Information Format, Version 3

7.9	ACCESSIBILITY CODES	148
7.10	VISIBILITY CODES	149
7.11	VIRTUALITY CODES	149
7.12	SOURCE LANGUAGES	149
7.13	ADDRESS CLASS ENCODINGS	151
7.14	IDENTIFIER CASE.....	151
7.15	CALLING CONVENTION ENCODINGS.....	152
7.16	INLINE CODES	152
7.17	ARRAY ORDERING	153
7.18	DISCRIMINANT LISTS	153
7.19	NAME LOOKUP TABLES.....	153
7.20	ADDRESS RANGE TABLE	154
7.21	LINE NUMBER INFORMATION	155
7.22	MACRO INFORMATION	157
7.23	CALL FRAME INFORMATION.....	157
7.24	NON-CONTIGUOUS ADDRESS RANGES.....	159
7.25	DEPENDENCIES AND CONSTRAINTS	160
7.26	INTEGER REPRESENTATION NAMES.....	161
APPENDIX A -- ATTRIBUTES BY TAG VALUE (INFORMATIVE).....		163
APPENDIX B -- DEBUG SECTION RELATIONSHIPS (INFORMATIVE).....		181
APPENDIX C -- VARIABLE LENGTH DATA: ENCODING/DECODING (INFORMATIVE).....		185
APPENDIX D -- EXAMPLES (INFORMATIVE)		187
D.1	COMPILATION UNITS AND ABBREVIATIONS TABLE EXAMPLE	187
D.2	AGGREGATE EXAMPLES	189
D.2.1	FORTRAN 90 EXAMPLE	189
D.2.2	ADA EXAMPLE	195
D.3	NAMESPACE EXAMPLES	198
D.4	MEMBER FUNCTION EXAMPLE.....	201
D.5	LINE NUMBER PROGRAM EXAMPLE	203
D.6	CALL FRAME INFORMATION EXAMPLE	205
D.7	INLINING EXAMPLES	209
D.7.1	ALTERNATIVE #1: INLINE BOTH OUTER AND INNER.....	210
D.7.2	ALTERNATIVE #2: INLINE OUTER, MULTIPLE INNERES	213
D.7.3	ALTERNATIVE #3: INLINE OUTER, ONE NORMAL INNER.....	216
APPENDIX E -- DWARF COMPRESSION AND DUPLICATE ELIMINATION (INFORMATIVE).....		219
E.1	OVERVIEW	219
E.2	NAMING AND USAGE CONSIDERATIONS.....	221
E.2.1	SECTION GROUP NAMES	222
E.2.2	DEBUGGING INFORMATION ENTRY NAMES.....	222
E.2.3	USE OF DW_TAG_COMPILE_UNIT VERSUS DW_TAG_PARTIAL_UNIT	223
E.2.4	USE OF DW_TAG_IMPORTED_UNIT	224
E.2.5	USE OF DW_FORM_REF_ADDR	224
E.3	EXAMPLES.....	225

E.3.1	C++ EXAMPLE	225
E.3.2	FORTRAN EXAMPLE	228
E.3.3	C EXAMPLE.....	231
E.4	SUMMARY OF COMPRESSION TECHNIQUES	232
E.4.1	#INCLUDE COMPRESSION	232
E.4.2	ELIMINATING FUNCTION DUPLICATION	232
E.4.3	SINGLE-FUNCTION-PER-DWARF-COMPILATION-UNIT	232
E.4.4	INLINING AND OUT-OF-LINE-INSTANCES.....	233
APPENDIX F – VERSION NUMBERS (INFORMATIVE)		235

DWARF Debugging Information Format, Version 3

List of Figures

FIGURE 1. TAG NAMES	6
FIGURE 2. ATTRIBUTE NAMES	11
FIGURE 3. CLASSES OF ATTRIBUTE VALUE	12
FIGURE 4. ACCESSIBILITY CODES	28
FIGURE 5. VISIBILITY CODES	28
FIGURE 6. VIRTUALITY CODES	29
FIGURE 7. EXAMPLE ADDRESS CLASS CODES	30
FIGURE 10. CALLING CONVENTION CODES	46
FIGURE 11. INLINE CODES	51
FIGURE 12. ENDIANITY ATTRIBUTE VALUES	62
FIGURE 13. ENCODING ATTRIBUTE VALUES	64
FIGURE 14. DECIMAL SIGN ATTRIBUTE VALUES	66
FIGURE 15. TYPE MODIFIER TAGS	68
FIGURE 16. ARRAY ORDERING	69
FIGURE 17. DISCRIMINANT DESCRIPTOR VALUES	79
FIGURE 18. TAG ENCODINGS	133
FIGURE 19. CHILD DETERMINATION ENCODINGS	133
FIGURE 20. ATTRIBUTE ENCODINGS	138
FIGURE 21. ATTRIBUTE FORM ENCODINGS	139
FIGURE 22. EXAMPLES OF UNSIGNED LEB128 ENCODINGS	140
FIGURE 23. EXAMPLES OF SIGNED LEB128 ENCODINGS	141
FIGURE 24. DWARF OPERATION ENCODINGS	145
FIGURE 25. BASE TYPE ENCODING VALUES	147
FIGURE 26. DECIMAL SIGN ENCODINGS	147
FIGURE 27. ENDIANITY ENCODINGS	148
FIGURE 28. ACCESSIBILITY ENCODINGS	148
FIGURE 29. VISIBILITY ENCODINGS	149
FIGURE 30. VIRTUALITY ENCODINGS	149
FIGURE 31. LANGUAGE ENCODINGS	151
FIGURE 32. IDENTIFIER CASE ENCODINGS	151
FIGURE 33. CALLING CONVENTION ENCODINGS	152
FIGURE 34. INLINE ENCODINGS	152
FIGURE 35. ORDERING ENCODINGS	153
FIGURE 36. DISCRIMINANT DESCRIPTOR ENCODINGS	153
FIGURE 37. LINE NUMBER STANDARD OPCODE ENCODINGS	156
FIGURE 38. LINE NUMBER EXTENDED OPCODE ENCODINGS	156
FIGURE 39. MACINFO TYPE ENCODINGS	157
FIGURE 40. CALL FRAME INSTRUCTION ENCODINGS	159
FIGURE 41. INTEGER REPRESENTATION NAMES	161
FIGURE 42. ATTRIBUTES BY TAG VALUE	179
FIGURE 43. DEBUG SECTION RELATIONSHIPS	182
FIGURE 44. ALGORITHM TO ENCODE AN UNSIGNED INTEGER	185
FIGURE 45. ALGORITHM TO ENCODE A SIGNED INTEGER	185
FIGURE 46. ALGORITHM TO DECODE AN UNSIGNED LEB128 NUMBER	186
FIGURE 47. ALGORITHM TO DECODE A SIGNED LEB128 NUMBER	186
FIGURE 48. COMPILATION UNITS AND ABBREVIATIONS TABLE	188

DWARF Debugging Information Format, Version 3

FIGURE 49. FORTRAN 90 EXAMPLE: SOURCE FRAGMENT	189
FIGURE 50. FORTRAN 90 EXAMPLE: DESCRIPTOR REPRESENTATION	190
FIGURE 51. FORTRAN 90 EXAMPLE: DWARF DESCRIPTION	193
FIGURE 52. ADA EXAMPLE: SOURCE FRAGMENT	195
FIGURE 53. ADA EXAMPLE: DWARF DESCRIPTION	197
FIGURE 54. NAMESPACE EXAMPLE: SOURCE FRAGMENT	198
FIGURE 55. NAMESPACE EXAMPLE: DWARF DESCRIPTION	200
FIGURE 56. MEMBER FUNCTION EXAMPLE: SOURCE FRAGMENT	201
FIGURE 57. MEMBER FUNCTION EXAMPLE: DWARF DESCRIPTION	202
FIGURE 58. LINE NUMBER PROGRAM EXAMPLE: MACHINE CODE	203
FIGURE 59. LINE NUMBER PROGRAM EXAMPLE: ONE ENCODING	204
FIGURE 60. LINE NUMBER PROGRAM EXAMPLE: ALTERNATE ENCODING	204
FIGURE 61. CALL FRAME INFORMATION EXAMPLE: MACHINE CODE FRAGMENTS	206
FIGURE 62. CALL FRAME INFORMATION EXAMPLE: CONCEPTUAL MATRIX	206
FIGURE 63. CALL FRAME INFORMATION EXAMPLE: COMMON INFORMATION ENTRY ENCODING	207
FIGURE 64. CALL FRAME INFORMATION EXAMPLE: FRAME DESCRIPTION ENTRY ENCODING	208
FIGURE 65. INLINING EXAMPLES: PSEUDO-SOURCE FRAGMENT	209
FIGURE 66. INLINING EXAMPLE #1: ABSTRACT INSTANCE	211
FIGURE 67. INLINING EXAMPLE #1: CONCRETE INSTANCE	212
FIGURE 68. INLINING EXAMPLE #2: ABSTRACT INSTANCE	214
FIGURE 69. INLINING EXAMPLE #2: CONCRETE INSTANCE	215
FIGURE 70. INLINING EXAMPLE #3: ABSTRACT INSTANCE	217
FIGURE 71. INLINING EXAMPLE #3: CONCRETE INSTANCE	218
FIGURE 72. DUPLICATE ELIMINATION EXAMPLE #1: C++ SOURCE	225
FIGURE 73. DUPLICATE ELIMINATION EXAMPLE #1: DWARF SECTION GROUP	226
FIGURE 74. DUPLICATE ELIMINATION EXAMPLE #1: PRIMARY COMPILATION UNIT	227
FIGURE 75. DUPLICATE ELIMINATION EXAMPLE #2: FORTRAN SOURCE	228
FIGURE 76. DUPLICATE ELIMINATION EXAMPLE #2: DWARF SECTION GROUP	229
FIGURE 77. DUPLICATE ELIMINATION EXAMPLE #2: PRIMARY UNIT	230
FIGURE 78. DUPLICATE ELIMINATION EXAMPLE #2: COMPANION SOURCE	230
FIGURE 79. DUPLICATE ELIMINATION EXAMPLE #2: COMPANION DWARF	231
FIGURE 80. SECTION VERSION NUMBERS	235

FOREWORD

The DWARF Version 3 specification is very similar to the DWARF Version 2 specification developed by the Programming Languages Special Interest Group (PLSIG) of UNIX International. Almost all DWARF Version 2 constructs have been retained unchanged in DWARF Version 3. It is the intention of the DWARF committee that migrating from DWARF Version 2 to DWARF Version 3 should be straightforward and easily accomplished. For further information, see <http://dwarf.freestandards.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 what are commonly called "hyperlinks" or "hot links"; these facilitate moving around in the document in a manner like that typically found in web browsers. 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 indistinguishable from other text. (Hyperlinks are not underlined for this same reason.) Page numbers, a Table of Contents and an Index appropriate to a paper document are available in both forms.

DWARF Debugging Information Format, Version 3

DWARF Debugging Information Format, Version 3

1 INTRODUCTION

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

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

1.1 Purpose and Scope

The debugging information format described in this document is designed to meet the symbolic, source-level debugging needs of different languages in a unified fashion by requiring language independent debugging information whenever possible. Individual needs, such as C++ virtual functions or Fortran common blocks are accommodated by creating attributes that are used only for those languages. This document is believed to cover most debugging information needs of C, C++, Fortran, Modula2 and Pascal; it also covers the basic needs of various other languages.

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

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

1.2 Overview

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

The informational content is described in sections two through six. Section two describes the overall structure of the information and attributes that is common to many or all of the different

DWARF Debugging Information Format, Version 3

debugging information entries. Sections three, four and five describe the specific debugging information entries and how they communicate the necessary information about the source program to a debugger. Section six describes debugging information contained outside of the debugging information entries. The encoding of the DWARF information is presented in Section seven.

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

In the following sections, text in normal font describes required aspects of the DWARF format. Text in *italics* is explanatory or supplementary material, and not part of the format definition itself. The several Appendices all consist only of explanatory or supplementary material, and are not part of the formal definition.

1.3 Vendor Extensibility

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

DWARF Version 3 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 backward compatible with this specification in the absence of those extensions.

1.4 Changes from Version 1 to Version 2

DWARF Version 2 describes the second generation of debugging information based on the DWARF format. While DWARF Version 2 provides new debugging information not available in Version 1, the primary focus of the changes for Version 2 is the representation of the information, rather than the information content itself. The basic structure of the Version 2 format remains as in Version 1: the debugging information is represented as a series of

INTRODUCTION

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.

1.5 Changes from Version 2 to Version 3

The following is a list of the major changes made to the DWARF Debugging Information Format since Version 2 of the format was published. 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.
- 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 Debugging Information Format, Version 3

1.5.1 Upward Compatibility

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

- Certain very large values of the initial length fields that begin DWARF sections as well as certain structures are reserved to act as escape codes for future extension; one such extension is defined to increase the possible size of DWARF descriptions (see Section [7.4](#)).
- References that use the attribute form `DW_FORM_ref_addr` are specified to be four bytes in the DWARF 32-bit format and eight bytes in the DWARF 64-bit format, while DWARF Version 2 specifies that such references have the same size as an address on the target system (see Sections [7.4](#) and [7.5.4](#)).
- 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](#)).

2 GENERAL DESCRIPTION

2.1 The Debugging Information Entry

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

The set of required tag names is listed in [Figure 1](#). The debugging information entries they identify are described in Sections three, four and five.

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

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

2.2 Attribute Types

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

The defined attribute names and their related uses are listed in [Figure 2](#).

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

Attribute value forms may belong to one of the classes shown in [Figure 3](#).

DWARF Debugging Information Format, Version 3

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

Figure 1. Tag names

GENERAL DESCRIPTION

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)
DW_AT_byte_size	Data object or data type size
DW_AT_byte_stride	Array dimension stride (from enumeration) Array dimension stride (from subrange)
DW_AT_call_column	Column position of inlined subroutine call
DW_AT_call_file	File containing inlined subroutine call
DW_AT_call_line	Line number of inlined subroutine call
DW_AT_calling_convention	Subprogram calling convention

DWARF Debugging Information Format, Version 3

Attribute	Identifies or Specifies
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_containing_type	Containing type of pointer to member type
DW_AT_count	Elements of subrange type
DW_AT_data_location	Indirection to actual data
DW_AT_data_member_location	Data member location Inherited member location
DW_AT_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
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

GENERAL DESCRIPTION

Attribute	Identifies or Specifies
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_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
DW_AT_inline	Abstract instance Inlined subroutine
DW_AT_is_optional	Optional parameter
DW_AT_language	Programming language
DW_AT_location	Data object location
DW_AT_low_pc	Code address or range of addresses
DW_AT_lower_bound	Lower bound of subrange

DWARF Debugging Information Format, Version 3

Attribute	Identifies or Specifies
DW_AT_macro_info	Macro information (#define, #undef)
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 (<i>this</i> , <i>self</i>) pointer of member function
DW_AT_ordering	Array row/column ordering
DW_AT_picture_string	Picture string for numeric string type
DW_AT_priority	Module priority
DW_AT_producer	Compiler identification
DW_AT_prototyped	Subroutine prototype
DW_AT_pure	Pure property of a subroutine
DW_AT_ranges	Non-contiguous range of code addresses
DW_AT_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_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

GENERAL DESCRIPTION

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

Figure 2. Attribute names

DWARF Debugging Information Format, Version 3

Attribute Class	General Use and Encoding
address	Refers to some location in the address space of the described program.
block	An arbitrary number of uninterpreted bytes of data.
constant	One, two, four or eight bytes of uninterpreted data, or data encoded in the variable length format known as LEB128 (see Section 7.6.). <i>Most constant values are integers of one kind or another (codes, offsets, counts, and so on); these are sometimes called “integer constants” for emphasis.</i>
flag	A small constant that indicates the presence or absence of an attribute.
lineptr	Refers to a location in the DWARF section that holds line number information.
loclistptr	Refers to a location in the DWARF section that holds location lists, which describe objects whose location can change during their lifetime.
macptr	Refers to a location in the DWARF section that holds macro definition information.
rangelistptr	Refers to a location in the DWARF section that holds non-contiguous address ranges.
reference	Refers to one of the debugging information entries that describe the program. There are two types of reference. The first is an offset relative to the beginning of the compilation unit in which the reference occurs and must refer to an entry within that same compilation unit. The second type of reference is the offset of a debugging information entry in any compilation unit, including one different from the unit containing the reference.
string	A null-terminated sequence of zero or more (non-null) bytes. Data in this form are generally printable strings. Strings may be represented directly in the debugging information entry or as an offset in a separate string table.

Figure 3. Classes of Attribute Value

GENERAL DESCRIPTION

2.3 Relationship of Debugging Information Entries

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

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

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

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

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

2.4 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.

DWARF Debugging Information Format, Version 3

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, additional register name operations (which are specific to [location expressions](#)) are defined in Section 2.6.1.

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

2.5.1 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, and so on).

2.5.1.1 Literal Encodings

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

1. **DW_OP_lit0, DW_OP_lit1, ..., DW_OP_lit31**

The DW_OP_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.

GENERAL DESCRIPTION

3. **DW_OP_const1u**
The single operand of the DW_OP_const1u operation provides a 1-byte unsigned integer constant.
4. **DW_OP_const1s**
The single operand of the DW_OP_const1s operation provides a 1-byte signed integer constant.
5. **DW_OP_const2u**
The single operand of the DW_OP_const2u operation provides a 2-byte unsigned integer constant.
6. **DW_OP_const2s**
The single operand of the DW_OP_const2s operation provides a 2-byte signed integer constant.
7. **DW_OP_const4u**
The single operand of the DW_OP_const4u operation provides a 4-byte unsigned integer constant.
8. **DW_OP_const4s**
The single operand of the DW_OP_const4s operation provides a 4-byte signed integer constant.
9. **DW_OP_const8u**
The single operand of the DW_OP_const8u operation provides an 8-byte unsigned integer constant.
10. **DW_OP_const8s**
The single operand of the DW_OP_const8s operation provides an 8-byte signed integer constant.
11. **DW_OP_constu**
The single operand of the DW_OP_constu operation provides an unsigned LEB128 integer constant.
12. **DW_OP_consts**
The single operand of the DW_OP_consts operation provides a signed LEB128 integer constant.

DWARF Debugging Information Format, Version 3

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 with a given signed offset.

1. **DW_OP_fbreg**

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

2. **DW_OP_breg0, DW_OP_breg1, ..., DW_OP_breg31**

The single operand of the DW_OP_breg*n* operations provides a signed LEB128 offset from the specified register.

3. **DW_OP_bregx**

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

2.5.1.3 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. The stack entry with the specified index (0 through 255, inclusive) is pushed on the stack.

4. **DW_OP_over**

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

5. **DW_OP_swap**

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

GENERAL DESCRIPTION

6. **DW_OP_rot**

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

7. **DW_OP_deref**

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

8. **DW_OP_deref_size**

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

9. **DW_OP_xderef**

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

10. **DW_OP_xderef_size**

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

DWARF Debugging Information Format, Version 3

11. DW_OP_push_object_address

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

This operator provides 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 dynamically loaded 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 expressions. 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.

GENERAL DESCRIPTION

2.5.1.4 Arithmetic and Logical Operations

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

1. **DW_OP_abs**
The DW_OP_abs operation pops the top stack entry, interprets it as a signed value and pushes its absolute value. If the absolute value can not be represented, the result is undefined.
2. **DW_OP_and**
The DW_OP_and operation pops the top two stack values, performs a bitwise *and* operation on the two, and pushes the result.
3. **DW_OP_div**
The DW_OP_div operation pops the top two stack values, divides the former second entry by the former top of the stack using signed division, and pushes the result.
4. **DW_OP_minus**
The DW_OP_minus operation pops the top two stack values, subtracts the former top of the stack from the former second entry, and pushes the result.
5. **DW_OP_mod**
The DW_OP_mod operation pops the top two stack values and pushes the result of the calculation: former second stack entry modulo the former top of the stack.
6. **DW_OP_mul**
The DW_OP_mul operation pops the top two stack entries, multiplies them together, and pushes the result.
7. **DW_OP_neg**
The DW_OP_neg operation pops the top stack entry, and pushes its negation.
8. **DW_OP_not**
The DW_OP_not operation pops the top stack entry, and pushes its bitwise complement.
9. **DW_OP_or**
The DW_OP_or operation pops the top two stack entries, performs a bitwise *or* operation on the two, and pushes the result.
10. **DW_OP_plus**
The DW_OP_plus operation pops the top two stack entries, adds them together, and pushes the result.

DWARF Debugging Information Format, Version 3

11. **DW_OP_plus_uconst**

The DW_OP_plus_uconst operation pops the top stack entry, adds it to the unsigned LEB128 constant operand and pushes the result.

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

12. **DW_OP_shl**

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

13. **DW_OP_shr**

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

14. **DW_OP_shra**

The DW_OP_shra operation pops the top two stack entries, shifts the former second entry right arithmetically (divide the magnitude by 2, keep the same sign for the result) by the number of bits specified by the former top of the stack, and pushes the result.

15. **DW_OP_xor**

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

2.5.1.5 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.

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

GENERAL DESCRIPTION

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

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

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

2.5.1.6 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.

DWARF Debugging Information Format, Version 3

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 1 29 2 1000	DW_OP_dup	0 17 1 17 2 29 3 1000
0 17 1 29 2 1000	DW_OP_drop	0 29 1 1000
0 17 1 29 2 1000	DW_OP_pick 2	0 1000 1 17 2 29 3 1000
0 17 1 29 2 1000	DW_OP_over	0 29 1 17 2 29 3 1000
0 17 1 29 2 1000	DW_OP_swap	0 29 1 17 2 1000
0 17 1 29 2 1000	DW_OP_rot	0 29 1 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.

GENERAL DESCRIPTION

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

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

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

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

2.6.1 Register Name Operators

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

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

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

2.6.2 Simple Location Expressions

A simple location expression describes the location of one contiguous piece of a value. A simple location expression is either:

1. A non-empty DWARF expression, whose value is the address of the piece of the value in memory

DWARF Debugging Information Format, Version 3

2. A DWARF expression containing no operations, which represents a piece of an object that is present in the source code but not in the object code (perhaps due to optimization), or
3. A register name operator, appearing alone as a single opcode, without other DWARF expression operators, representing a piece of the object located in the given register.

2.6.3 Composition Operators

A composition operator describes the location of a piece of a value which may be contained in part of a register or stored in more than one location; it does not compute a value nor store any result on the DWARF stack. There may be one or more composition operators in a single DWARF location expression. A series of such operators describes the parts of a value in memory address order.

Each composition operator may be immediately preceded by a register name operator or an addressing operation which describes the location where part of the resultant value is contained.

1. **DW_OP_piece**

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

2. **DW_OP_bit_piece**

DW_OP_bit_piece 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 expression.

Interpretation of the offset depends on the kind of location expression. If the location is null, the offset doesn't matter and the **DW_OP_bit_piece** operator 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** operator 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.

***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.*

GENERAL DESCRIPTION

2.6.4 Location Expressions

A location expression is either:

1. A simple location expression, representing an object which exists in one piece at the given location, or
2. Two or more simple location expressions, each of which is followed by one composition operator. Each simple location expression describes the location of one piece of the object; each composition operator describes which part of the object is located there. Each simple location expression that is a DWARF expression is evaluated independently of any others (as though on its own separate stack).

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

2.6.5 Example Location Expressions

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

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

`DW_OP_reg3`

The value is in register 3.

`DW_OP_regx 54`

The value is in register 54.

`DW_OP_addr 0x80d0045c`

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

`DW_OP_breg11 44`

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

`DW_OP_fbreg -50`

Given an `DW_AT_frame_base` value of "`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).

DWARF Debugging Information Format, Version 3

`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.

2.6.6 Location Lists

Location lists are used in place of location expressions whenever the object whose location is being described can change location during its lifetime. Location lists are contained in a separate object file section called `.debug_loc`. A location list is indicated by a location attribute whose value is represented as 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 location expression describing the location of the object over the range specified by the beginning and end addresses.

GENERAL DESCRIPTION

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

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

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

A base address selection entry consists of:

1. The value of the largest representable address 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 expression.

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

2.7 Types of Declarations

Any debugging information entry describing a declaration that has a type has a [DW_AT_type](#) attribute, whose value is a reference to another debugging information entry. The entry referenced may describe a base type, that is, a type that is not defined in terms of other data types, or it may describe a user-defined type, such as an array, structure or enumeration. Alternatively, the entry referenced may describe a type modifier: constant, packed, pointer, reference or volatile, which in turn will reference another entry describing a type or type

DWARF Debugging Information Format, Version 3

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 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

GENERAL DESCRIPTION

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. The value of this attribute 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 value of the item being described. If the entry containing the [DW_AT_segment](#) attribute has a [DW_AT_low_pc](#), [DW_AT_high_pc](#), [DW_AT_ranges](#) or [DW_AT_entry_pc](#) attribute, or a location description that evaluates to an address, then those address values represent the offset portion of the address within the segment specified by [DW_AT_segment](#).

If an entry has no [DW_AT_segment](#) attribute, it inherits the segment value from its parent entry. If none of the entries in the chain of parents for this entry back to its containing compilation unit

DWARF Debugging Information Format, Version 3

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:

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

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.

GENERAL DESCRIPTION

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

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

2.13.2 Declarations Completing Non-Defining Declarations

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

2.14 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.

DWARF Debugging Information Format, Version 3

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. Sequences of multiple whitespace characters may be compressed.

2.16 Data Locations

Any debugging information entry describing a data object, which includes variables, parameters, common blocks and the like, may have a [DW_AT_location](#) attribute, whose value is a location description (see Section 2.6).

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.

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.

GENERAL DESCRIPTION

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 contiguous range, the entry may have a `DW_AT_low_pc` and `DW_AT_high_pc` pair of attributes. The value of the `DW_AT_low_pc` attribute is the relocated address of the first instruction associated with the entity, and the value of the `DW_AT_high_pc` is the relocated address of the first location past the last instruction associated with the entity.

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

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

2.17.3 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.

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.

DWARF Debugging Information Format, Version 3

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).

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 2.6.6) 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_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.

GENERAL DESCRIPTION

2.19 Static and Dynamic Properties of Types

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

- For a [constant](#), the value of the constant is the value of the attribute.
- For a [reference](#), the value is a reference to another entity whose value is the value of the attribute.
- For a [block](#), the value is interpreted as a [DWARF expression](#); evaluation of the expression yields the value of the attribute.

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

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

2.20 Entity Descriptions

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

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

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

DWARF Debugging Information Format, Version 3

3 PROGRAM SCOPE ENTRIES

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

3.1 Compilation and Importing Entries

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

3.1.1 Normal and Partial Compilation Unit Entries

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

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

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

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

DWARF Debugging Information Format, Version 3

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 2.6.6) 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
<code>DW_LANG_Ada83</code> †	ISO Ada:1983
<code>DW_LANG_Ada95</code> †	ISO Ada:1995
<code>DW_LANG_C</code>	Non-standardized C, such as K&R
<code>DW_LANG_C89</code>	ISO C:1989
<code>DW_LANG_C99</code>	ISO C:1999
<code>DW_LANG_C_plus_plus</code>	ISO C++:1998
<code>DW_LANG_Cobol74</code> †	ISO Cobol:1974
<code>DW_LANG_Cobol85</code> †	ISO Cobol:1985
<code>DW_LANG_D</code> †	D
<code>DW_LANG_Fortran77</code>	ISO FORTRAN 77
<code>DW_LANG_Fortran90</code>	ISO Fortran 90

PROGRAM SCOPE ENTRIES

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_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_macro` section of the first byte of the macro information for this compilation unit (see Section 6.3).

DWARF Debugging Information Format, Version 3

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

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

7. A `DW_AT_producer` attribute whose value is a null-terminated string containing information about the compiler that produced the compilation unit. The actual contents of the string will be specific to each producer, but should begin with the name of the compiler vendor or some other identifying character sequence that should avoid confusion with other producer values.
8. A `DW_AT_identifier_case` attribute whose `integer constant` value is a code describing the treatment of identifiers within this compilation unit. The set of identifier case codes is given in [Figure 9](#).

<code>DW_ID_case_sensitive</code>
<code>DW_ID_up_case</code>
<code>DW_ID_down_case</code>
<code>DW_ID_case_insensitive</code>

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.

PROGRAM SCOPE ENTRIES

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

9. A `DW_AT_base_types` attribute whose value is a [reference](#).

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

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

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

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

3.1.2 Imported Unit Entries

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

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

3.2 Module, Namespace and Importing Entries

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

3.2.1 Module Entries

Several languages have the concept of a “module.”

DWARF Debugging Information Format, Version 3

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.

A Modula2 definition module may be represented by a module entry containing a declaration attribute (`DW_AT_declaration`). A Fortran 90 module may also be represented by a module entry (but no declaration attribute is warranted because Fortran has no concept of a corresponding module body).

3.2.2 Namespace Entries

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

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

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

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

PROGRAM SCOPE ENTRIES

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. 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

DWARF Debugging Information Format, Version 3

different than the name of the entity being imported). If no name is present, then the name by which the entity is to be known is the same as the name of the entity being imported.

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

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

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

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

3.2.4 Imported Module Entries

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

An imported module declaration is represented by a debugging information entry with the tag DW_TAG_imported_module. An imported module entry contains a DW_AT_import attribute whose value is a reference to the module or namespace entry containing the definition and/or declaration entries for the entities that are to be imported into the context of the imported 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 referenced in this way is known in the context of the imported module entry by the same name as it is declared in the module.

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

PROGRAM SCOPE ENTRIES

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

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

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

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

module B
  use A
end module

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

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

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

3.3 Subroutine and Entry Point Entries

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

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

3.3.1 General Subroutine and Entry Point Information

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

DWARF Debugging Information Format, Version 3

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

Additional attributes for functions that are members of a class or structure are described in Section 5.6.7.

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

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

<code>DW_CC_normal</code>
<code>DW_CC_program</code>
<code>DW_CC_nocall</code>

Figure 10. Calling convention codes

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

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

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

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

PROGRAM SCOPE ENTRIES

A subprogram entry may have a [DW_AT_pure](#) attribute, whose value 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, whose value is a flag. The attribute indicates whether the subroutine or entry point was declared with the “recursive” keyword or property.

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

3.3.2 Subroutine and Entry Point Return Types

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

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

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

A subroutine entry declared with a function prototype style declaration may have a [DW_AT_prototyped](#) attribute whose value is a flag.

3.3.3 Subroutine and Entry Point Locations

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

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.

DWARF Debugging Information Format, Version 3

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 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 register operation, 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 expression contained in the list of location list entries.

PROGRAM SCOPE ENTRIES

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

- 1. In procedures that need location lists to locate local variables, the [DW_AT_frame_base](#) can hold the needed location list, while all variables' location descriptions can be simpler location expressions involving the frame base.*
- 2. It can be used as a key in resolving "up-level" addressing with nested routines. (See [DW_AT_static_link](#), below)*

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

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

In the context of supporting nested subroutines, the [DW_AT_frame_base](#) attribute value should obey the following constraints:

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

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

3.3.6 Types Thrown by Exceptions

In C++ a subroutine may declare a set of types 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. All thrown type entries should follow all entries representing the formal parameters of the subroutine and precede all entries representing the local variables or lexical blocks contained in the

DWARF Debugging Information Format, Version 3

subroutine. Each thrown type entry contains a [DW_AT_type](#) attribute, whose value is a [reference](#) to an entry describing the type of the exception that may be thrown.

3.3.7 Function Template Instantiations

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

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

1. Each formal parameterized type declaration appearing in the template definition is represented by a debugging information entry with the tag [DW_TAG_template_type_parameter](#). Each such entry has a [DW_AT_name](#) attribute, whose value is a null-terminated string containing the name of the formal type parameter as it appears in the source program. The template type parameter entry also has a [DW_AT_type](#) attribute describing the actual type by which the formal is replaced for this instantiation. All template type parameter entries should appear before the entries describing the instantiated formal parameters to the function.
2. If the compiler has generated a special compilation unit to hold the template instantiation and that compilation unit has a different name from the compilation unit containing the template definition, the name attribute for the debugging entry representing that compilation unit should be empty or omitted.
3. If the subprogram entry representing the template instantiation or any of its child entries contain [declaration coordinate](#) attributes, those attributes should refer to the source for the template definition, not to any source generated artificially by the compiler for this instantiation.

3.3.8 Inline Subroutines

A declaration or a definition of an inlinable subroutine is represented by a debugging information entry with the tag [DW_TAG_subprogram](#). The entry for a subroutine that is explicitly declared to be available for inline expansion or that was expanded inline implicitly by the compiler has a [DW_AT_inline](#) attribute whose value is an [integer constant](#). The set of values for the [DW_AT_inline](#) attribute is given in [Figure 11](#).

PROGRAM SCOPE ENTRIES

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

Figure 11. Inline codes

3.3.8.1 Abstract Instances

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

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

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

It would not make sense normally to put these attributes into abstract instance entries since such entries do not represent actual (concrete) instances and thus do not actually exist at run-time. However, see Appendix [D.7.3](#) for a contrary example.

DWARF Debugging Information Format, Version 3

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

3.3.8.2 Concrete Inlined Instances

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

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

Any debugging information entry that is owned (either directly or indirectly) by a debugging information entry with the tag `DW_TAG_inlined_subroutine` is referred to as a “concrete inlined instance entry.” Any entry that has the tag `DW_TAG_inlined_subroutine` is known as a “concrete inlined instance root.” Any set of concrete inlined instance entries that are all children (either directly or indirectly) of some concrete inlined instance root, together with the root itself, is known as a “concrete inlined instance tree.” However, in the case where a concrete instance tree is nested within another concrete instance tree, the entries in the nested concrete instance tree are not considered to be entries in the outer concrete instance tree.

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

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

PROGRAM SCOPE ENTRIES

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

Concrete inlined instance entries may omit attributes that are not specific to the concrete instance (but present in the abstract instance) and need include only attributes that are specific to the concrete instance (but omitted in the abstract instance). In place of these omitted attributes, each concrete inlined instance entry has a [DW_AT_abstract_origin](#) attribute that may be used to obtain the missing information (indirectly) from the associated abstract instance entry. The value of the abstract origin attribute is a [reference](#) to the associated abstract instance entry.

If an entry within a concrete inlined instance tree contains attributes describing the [declaration coordinates](#) of that entry, then those attributes should refer to the file, line and column of the original declaration of the subroutine, not to the point at which it was inlined. As a consequence, they may usually be omitted from any entry that has an abstract origin attribute.

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

In general, the structure and content of any given concrete inline instance tree will be closely analogous to the structure and content of its associated abstract instance tree. There are a few exceptions:

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

DWARF Debugging Information Format, Version 3

2. Entries in the concrete instance tree which are associated with entries in the abstract instance tree such that neither has a [DW_AT_name](#) attribute, and neither is referenced by any other debugging information entry, may be omitted. This may happen for debugging information entries in the abstract instance trees that became unnecessary in the concrete instance tree because of additional information available there. For example, an anonymous variable might have been created and described in the abstract instance tree, but because of the actual parameters for a particular inline expansion, it could be described as a constant value without the need for that separate debugging information entry.
3. A concrete instance tree may contain entries which do not correspond to entries in the abstract instance tree to describe new entities that are specific to a particular inline expansion. In that case, they will not have associated entries in the abstract instance tree, should not contain [DW_AT_abstract_origin](#) attributes, and must contain all their own attributes directly. This allows an abstract instance tree to omit debugging information entries for anonymous entities that are unlikely to be needed in most inline expansions. In any expansion which deviates from that expectation, the entries can be described in its concrete instance tree.

3.3.8.3 Out-of-Line Instances of Inline Subroutines

Under some conditions, compilers may need to generate concrete executable instances of inline subroutines other than at points where those subroutines are actually called. Such concrete instances of inline subroutines are referred to as “concrete out-of-line instances.”

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

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

PROGRAM SCOPE ENTRIES

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

1. The root entry for a concrete out-of-line instance of a given inline subroutine has the same tag as does its associated (abstract) inline subroutine entry (that is, it does not have the tag `DW_TAG_inlined_subroutine`).
2. The root entry for a concrete out-of-line instance tree is normally owned by the same parent entry that also owns the root entry of the associated abstract instance. However, it is not required that the abstract and out-of-line instance trees be owned by the same parent entry.

3.3.8.4 Nested Inline Subroutines

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

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

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

1. The abstract instance tree for the nested subroutine is described within the abstract instance tree for the outer subroutine according to the rules in Section 3.3.8.1, and without regard to the fact that it is within an outer abstract instance tree.
2. Any abstract instance tree for a nested subroutine is always omitted within the concrete instance tree for an outer subroutine.
3. A concrete instance tree for a nested subroutine is always omitted within the abstract instance tree for an outer subroutine.
4. The concrete instance tree for any inline or out-of-line expansion of the nested subroutine is described within a concrete instance tree for the outer subroutine according to the rules in Sections 3.3.8.2 or 3.3.8.3, respectively, and without regard to the fact that it is within an outer concrete instance tree.

DWARF Debugging Information Format, Version 3

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.

PROGRAM SCOPE ENTRIES

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.

DWARF Debugging Information Format, Version 3

3.6 With Statement Entries

Both Pascal and Modula2 support the concept of a “with” statement. The with statement specifies a sequence of executable statements within which the fields of a record variable may be referenced, unqualified by the name of the record variable.

A with statement is represented by a debugging information entry with the tag `DW_TAG_with_stmt`.

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

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 will have one of the tags `DW_TAG_formal_parameter` or `DW_TAG_unspecified_parameters`, and will have the same form as other parameter entries.

The siblings following any try block entry will be 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 a C++ anonymous union, the name attribute is omitted or consists of a single zero byte.

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

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

3. A `DW_AT_declaration` attribute, whose value is a flag, which indicates 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 a null 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).

DWARF Debugging Information Format, Version 3

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

The location of a variable may be further specified with a [DW_AT_segment](#) attribute, if appropriate.

5. A [DW_AT_type](#) attribute describing the type of the variable, constant or formal parameter.
6. If the variable entry represents the defining declaration for a C++ static data member of a structure, class or union, the entry has a [DW_AT_specification](#) attribute, whose value is a [reference](#) to the debugging information entry representing the declaration of this data member. The referenced entry has the tag [DW_TAG_member](#) and will be a child of some class, structure or union type entry.

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

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

7. If a formal parameter entry represents a parameter whose value in the calling function may be modified by the callee, that entry may have a [DW_AT_variable_parameter](#) attribute, whose value is a [flag](#). The absence of this attribute implies that the parameter's value in the calling function cannot be modified by the callee.
8. If a parameter entry represents an optional parameter, it has a [DW_AT_is_optional](#) attribute, whose value is a [flag](#).
9. A formal parameter entry may have a [DW_AT_default_value](#) attribute. The value of this attribute is a [reference](#) to the debugging information entry for a variable or subroutine, or the value may be a constant. If it is a reference, the default value of the parameter is the value of the variable (which may be constant) or the value returned by the subroutine. If the value of the [DW_AT_default_value](#) attribute is 0, it means that no default value has been specified. If the value is of form constant, that constant is interpreted as a value of the type of the formal parameter.

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

DATA OBJECT AND OBJECT LIST ENTRIES

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

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

11. If the scope of an object begins sometime after the low pc value for the scope most closely enclosing the object, the object entry may have a [DW_AT_start_scope](#) attribute. 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.

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 .

12. A [DW_AT_endianity](#) attribute whose value is a constant that specifies the endianness of the object. The value of this attribute specifies an ABI-defined byte ordering for the value of the object. If omitted, or the value for this attribute is zero, the default endianness of data for the given type is assumed.

DWARF Debugging Information Format, Version 3

The set of values and their meaning for this attribute is given in [Figure 12](#).

Name	Meaning
DW_END_default	Default endian encoding
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.

4.2 Common Block Entries

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

4.3 Namelist Entries

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

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

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

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

5 TYPE ENTRIES

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

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

5.1 Base Type Entries

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

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

A base type entry also has a [DW_AT_encoding](#) attribute describing how the base type is encoded and is to be interpreted. The value of this attribute is an [integer constant](#). The set of values and their meanings for the [DW_AT_encoding](#) attribute is given in [Figure 13](#) and following text.

A base type entry may also 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 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.

DWARF Debugging Information Format, Version 3

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.

Name	Meaning
DW_ATE_address	linear machine address
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

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.

TYPE ENTRIES

The `DW_ATE_packed_decimal` and `DW_ATE_numeric_string` base types represent decimal string numeric data types; packed and unpacked, and, signed and 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 a COBOL 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 are interpreted in the context of 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.

DWARF Debugging Information Format, Version 3

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 an arbitrary scale factor, the fixed binary type entry has a `DW_AT_small` attribute. It references a `DW_TAG_constant` entry with an arbitrary type attribute. The scale factor value is interpreted in accordance with the value enumerated in 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
<code>DW_DS_unsigned</code>	Unsigned
<code>DW_DS_leading_overpunch</code>	Sign is encoded in the most significant digit in a target-dependent manner
<code>DW_DS_trailing_overpunch</code>	Sign is encoded in the least significant digit in a target-dependent manner
<code>DW_DS_leading_separate</code>	Sign is a '+' or '-' character to the left of the most significant digit
<code>DW_DS_trailing_separate</code>	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

TYPE ENTRIES

5.2 Unspecified Type Entries

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

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

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

5.3 Type Modifier Entries

A base or user-defined type may be modified in different ways in different languages. A type modifier is represented in DWARF by a debugging information entry with one of the tags given in [Figure 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 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 (see [Section 5.12](#)) 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.

DWARF Debugging Information Format, Version 3

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

Tag	Meaning
DW_TAG_const_type	C or C++ const qualified type
DW_TAG_packed_type	Pascal or Ada packed type
DW_TAG_pointer_type	Pointer to an object of the type being modified.
DW_TAG_reference_type	C++ reference to an object of the type being modified
DW_TAG_restrict_type	C restrict qualified type
DW_TAG_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)
```

TYPE ENTRIES

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

5.4 Typedef Entries

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

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

5.5 Array Type Entries

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

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

The array type entry describing a multidimensional array may have a `DW_AT_ordering` attribute whose `integer constant` value is interpreted to mean either row-major or column-major ordering of array elements. The set of values and their meanings for the ordering attribute are listed in [Figure 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

DWARF Debugging Information Format, Version 3

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

An array type entry has a [DW_AT_type](#) attribute describing the type of each element of the array.

If the amount of storage allocated to hold each element of an object of the given array type is different from the amount of storage that is normally allocated to hold an individual object of the indicated element type, then the array type entry has a [DW_AT_bit_stride](#) attribute, whose value (see Section 2.19) is the size in bits of each element of the array.

The name [DW_AT_bit_stride](#) replaces the name [DW_AT_stride_size](#) which is defined in DWARF Version 2. The name [DW_AT_stride_size](#) may be used as an alias for [DW_AT_bit_stride](#) but use of [DW_AT_stride_size](#) is deprecated.

The array type entry may have a [DW_AT_byte_size](#) attribute, whose value (see Section 2.19) is the total size in bytes of an instance of the array type.

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

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

In languages, such as 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 5.15. For relevant examples, see also Appendix D.2.1.

5.6 Structure, Union, Class and Interface Type Entries

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

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

TYPE ENTRIES

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

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

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

5.6.1 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.

If the size of an instance of the structure type, union type, or class type entry can be determined statically at compile time, the entry has a DW_AT_byte_size attribute whose integer constant value is the number of bytes required to hold an instance of the structure, union, or class, including any padding bytes. Otherwise, the entry may have a DW_AT_byte_size attribute whose value (see Section 2.19) is the dynamic number of bytes required.

An incomplete structure, union or class type is represented by a structure, union or class entry that does not have a byte size attribute and that has a DW_AT_declaration attribute.

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

Structure, union and class entries containing the DW_AT_specification attribute do not need to duplicate information provided by the declaration entry referenced by the specification attribute. In particular, such entries do not need to contain an attribute for the name of structure, class or union they represent.

DWARF Debugging Information Format, Version 3

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

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

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

```
struct A {  
    struct B;  
};  
  
struct A::B { ... };
```

The two different structs can be described in different compilation units to facilitate DWARF space compression (see Appendix E).

5.6.2 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.

TYPE ENTRIES

5.6.3 Derived or Extended Classes and Interfaces

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

A class type or interface type entry that describes a derived, extended or implementing class or interface owns debugging information entries describing each of the classes or interfaces it is derived from, extending or implementing, respectively, ordered as they were in the source program. Each such entry has the tag `DW_TAG_inheritance`.

An inheritance entry has a `DW_AT_type` attribute whose value is a [reference](#) to the debugging information entry describing the class or interface from which the parent class or structure of the inheritance entry is derived, extended or implementing.

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

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

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

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

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

5.6.4 Access Declarations

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

DWARF Debugging Information Format, Version 3

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

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

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

5.6.5 Friends

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

A friend entry has a `DW_AT_friend` attribute, whose value is a [reference](#) to the debugging information entry describing the declaration of the friend.

5.6.6 Data Member Entries

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

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

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

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

If a data member is defined in a structure, union or class, the corresponding member entry has a `DW_AT_data_member_location` attribute whose value describes the location of that member relative to the base address of the structure, union, or class that most closely encloses the member declaration. If that value is a constant, it is the offset in bytes from the beginning of the innermost enclosing structure, union or class to the beginning of the data member. Otherwise, the value must be a [location description](#). In this latter case, the base address of the innermost

TYPE ENTRIES

enclosing structure, union, or class is pushed on the expression stack before the location expression is evaluated and the result of the evaluation is the location of the member.

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

For a location description, the implicit push on the DWARF expression stack of the base address of the containing construct is equivalent to execution of the [DW_OP_push_object_address](#) operation (see Section 2.5.1.3); [DW_OP_push_object_address](#) therefore is not needed at the beginning of a location expression for a data member. The result of the evaluation is a location--either an address or the name of a register, not an offset to the member.

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

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

- A [DW_AT_byte_size](#) attribute whose value (see Section 2.19) 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.

DWARF Debugging Information Format, Version 3

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

```

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

In both cases, the location descriptions for the debugging information entries for *j*, *k*, *m* and *n* describe the address of the same 32-bit word that contains all three members. (In the big-endian case, the location description addresses the most significant byte, in the little-endian case, the least significant).

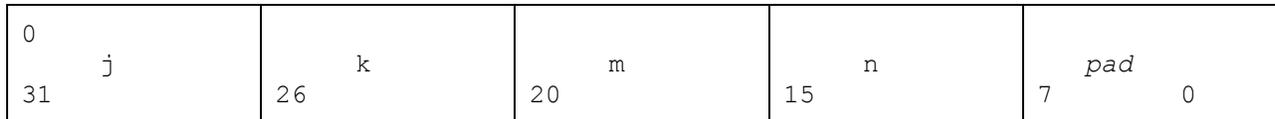
The following diagram shows the structure layout and lists the bit offsets for each case. The offsets are from the most significant bit of the object addressed by the location description.

Bit Offsets:

```

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

Big-Endian

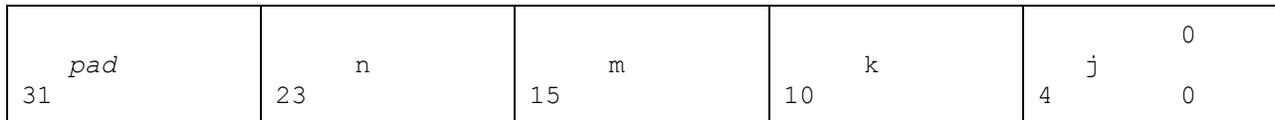


Bit Offsets:

```

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

Little-Endian



TYPE ENTRIES

5.6.7 Member Function Entries

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

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

If the member function entry describes a virtual function, then that entry has a [DW_AT_virtuality](#) attribute.

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.

The type of the formal parameter of a non-static member function need not be checked by a consumer to confirm that it is a pointer to the parent class or a const-volatile qualified variant thereof because the requirements of correct DWARF make such a test superfluous.

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

DWARF Debugging Information Format, Version 3

representing the declaration of this function member. The referenced entry will be a child of some class (or structure) type entry.

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

5.6.8 Class Template Instantiations

In C++ a class template is a generic definition of a class type that 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 four exceptions, such an entry will contain the same attributes and have the same types of child entries as would an entry for a class type defined explicitly using the instantiation types and values. The exceptions are:

1. Each formal parameterized type declaration appearing in the template definition is represented by a debugging information entry with the tag [DW_TAG_template_type_parameter](#). Each such entry may have a [DW_AT_name](#) attribute, whose value is a null-terminated string containing the name of the formal type parameter as it appears in the source program. The template type parameter entry also has a [DW_AT_type](#) attribute describing the actual type by which the formal is replaced for this instantiation.
2. Each formal parameterized value declaration appearing in the template definition is represented by a debugging information entry with the tag [DW_TAG_template_value_parameter](#). Each such entry may have a [DW_AT_name](#) attribute, whose value is a null-terminated string containing the name of the formal value parameter as it appears in the source program. The template value parameter entry also has a [DW_AT_type](#) attribute describing the type of the parameterized value. Finally, the template value parameter entry has a [DW_AT_const_value](#) attribute, whose value is the actual constant value of the value parameter for this instantiation as represented on the target architecture.
3. If the compiler has generated a special compilation unit to hold the template instantiation and that compilation unit has a different name from the compilation unit containing the template definition, the name attribute for the debugging entry representing that compilation unit should be empty or omitted.

TYPE ENTRIES

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

5.6.9 Variant Entries

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

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

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

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

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

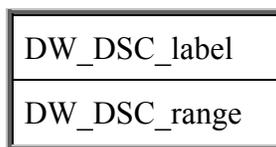


Figure 17. Discriminant descriptor values

DWARF Debugging Information Format, Version 3

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

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

5.7 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 COBOL level-88 condition. 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](#) or [DW_TAG_member](#) entry. The type of the condition entry is derived from the [DW_AT_type](#) attribute 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 type derived from the condition entry's parent.

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.

5.8 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](#).

TYPE ENTRIES

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 also have a [DW_AT_type](#) attribute which refers to the underlying data type used to implement the enumeration.

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.

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.

The name [DW_AT_byte_stride](#) replaces the name [DW_AT_stride](#) which is defined in DWARF Version 2. The name [DW_AT_stride](#) may be used as an alias for [DW_AT_byte_stride](#) but use of [DW_AT_stride](#) is deprecated.

5.9 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

DWARF Debugging Information Format, Version 3

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, whose value is a [flag](#).

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

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

5.10 String Type Entries

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

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

The string type entry may have a [DW_AT_string_length](#) attribute whose value is a location description yielding the location where the length of the string is stored in the program. The string type entry may also have a [DW_AT_byte_size](#) attribute, whose value (see Section 2.19) is the size in bytes of the data to be retrieved from the location referenced by the string length

TYPE ENTRIES

attribute. If no byte size attribute is present, the size of the data to be retrieved is the same as the size of an address on the target machine.

If no string length attribute is present, the string type entry may have a [DW_AT_byte_size](#) attribute, whose value (see Section 2.19) is the length in bytes of the string.

5.11 Set Entries

Pascal provides the concept of a “set,” which represents a group of values of ordinal type.

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

The set type entry has a [DW_AT_type](#) attribute to denote the type of an element of the set.

If the amount of storage allocated to hold each element of an object of the given set type is different from the amount of storage that is normally allocated to hold an individual object of the indicated element type, then the set type entry has a [DW_AT_byte_size](#) attribute, whose value (see Section 2.19) is the size in bytes of an instance of the set type.

5.12 Subrange Type Entries

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

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

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

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

The subrange entry may have a [DW_AT_threads_scaled](#) attribute whose value 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).

DWARF Debugging Information Format, Version 3

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 describe, respectively, the lower and upper bound values of the subrange. The [DW_AT_upper_bound](#) attribute may be replaced by a [DW_AT_count](#) attribute, whose value describes the number of elements in the subrange rather than the value of the last element. The value of each of these attributes is determined as described in Section 2.19.

If the lower bound value is missing, the value is assumed to be a language-dependent default constant. The default lower bound value for C or C++ is 0. For Fortran, it is 1.

No other default lower bound values are currently defined.

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

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

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

Note that the stride can be negative.

The name [DW_AT_byte_stride](#) replaces the name [DW_AT_stride](#) which is defined in DWARF Version 2. The name [DW_AT_stride](#) may be used as an alias for [DW_AT_byte_stride](#) but use of [DW_AT_stride](#) is deprecated.

5.13 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](#).

TYPE ENTRIES

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

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

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

Finally, the pointer to member entry has a [DW_AT_use_location](#) attribute whose value is a location description that computes the address of the member of the class to which the pointer to member entry points (the value is meaningless if the pointer to member does not currently point to anything).

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

The [DW_AT_use_location](#) expression must be used in conjunction with the location expressions for a particular object of the given pointer to member type and for a particular structure or class instance. The [DW_AT_use_location](#) attribute expects two values to be pushed onto the location expression stack before the [DW_AT_use_location](#) expression is evaluated. The first value pushed is the value of the pointer to member object itself. The second value pushed is the base address of the entire structure or union instance containing the member whose address is being calculated.

For an expression such as

*object.*mbr_ptr*

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

- 1. Push the value of mbr_ptr onto the location expression stack.*
- 2. Push the base address of object onto the location expression stack.*
- 3. Evaluate the [DW_AT_use_location](#) expression given in the type of mbr_ptr.*

DWARF Debugging Information Format, Version 3

5.14 File Type Entries

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

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

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

The file type entry also has a `DW_AT_byte_size` attribute, whose value (see Section 2.19) is the size in bytes of an instance of this file type.

5.15 Dynamic Type Properties

5.15.1 Data Location

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

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

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

5.15.2 Allocation and Association Status

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

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

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

TYPE ENTRIES

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](#).

5.16 DWARF Procedures

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

A DWARF procedure is called by a [DW_OP_call2](#), [DW_OP_call4](#) or [DW_OP_call_ref](#) DWARF expression operator (see Section [2.5.1.5](#)).

DWARF Debugging Information Format, Version 3

6 OTHER DEBUGGING INFORMATION

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

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

- Initial length, section offset and section length, which are defined in Sections 7.2.2 and 7.4.
- Sbyte, ubyte, uhalf, and uword, which are defined in Section 7.26.

6.1 Accelerated Access

A debugger frequently needs to find the debugging information for a program entity defined outside of the compilation unit where the debugged program is currently stopped. Sometimes it will know only the name of the entity; sometimes only the address. To find the debugging information associated with a global entity by name, using the DWARF debugging information entries alone, a debugger would need to run through all entries at the highest scope within each compilation unit.

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

For lookup by address, for a subroutine, a debugger can use the low and high pc attributes of the compilation unit entries to quickly narrow down the search, but these attributes only cover the range of addresses for the text associated with a compilation unit entry. To find the debugging information associated with a data object, an exhaustive search would be needed. Furthermore, any search through debugging information entries for different compilation units within a large program would potentially require the access of many memory pages, probably hurting debugger performance.

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

DWARF Debugging Information Format, Version 3

6.1.1 Lookup by Name

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

C++ member functions with a definition in the class declaration are definitions in every compilation unit containing the class declaration, but if there is no concrete 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 length 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 name lookup table and is independent of the DWARF version number.

3. `debug_info_offset` (section offset)

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

4. `debug_info_length` (section length)

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

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

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

OTHER DEBUGGING INFORMATION

6.1.2 Lookup by Address

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

Each set begins with a header containing five values:

1. `unit_length` (initial length)

The length of the entries for that set, not including the length field itself (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` 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 descriptor 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 pair consisting of the beginning address of a range of text or data covered by some entry owned by the corresponding compilation unit, followed by the non-zero length of that range. A particular set is terminated by an entry consisting of two zeroes. By scanning the table, a debugger can quickly decide which compilation unit to look in to find the debugging information for an object that has a given address.

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

DWARF Debugging Information Format, Version 3

6.2 Line Number Information

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

As mentioned in Section 3.1.1, the line number information generated for a compilation unit is represented in the `.debug_line` section of an object file and is referenced by a corresponding compilation unit debugging information entry in the `.debug_info` section.

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

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

- *the source file name*
- *the source line number*
- *the source column number*
- *whether this instruction is the beginning of a source statement*
- *whether this instruction is the beginning of a basic block*
- *and so on*

Such a matrix, however, would be impractically large. We shrink it with two techniques. First, we delete from the matrix each row whose file, line and source column information is identical with that of its predecessors. Any deleted row would never be the beginning of a source statement. Second, we design a byte-coded language for a state machine and store a stream of bytes in the object file instead of the matrix. This language can be much more compact than the matrix. When a consumer of the line number information executes, it must “run” the state machine to generate the matrix for each compilation unit it is interested in. The concept of an

OTHER DEBUGGING INFORMATION

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. <i>A basic block does not necessarily correspond to a specific source code construct.</i>
sequence	A series of contiguous target machine instructions. One compilation unit may emit multiple sequences (that is, not all instructions within a compilation unit are assumed to be contiguous).

6.2.2 State Machine Registers

The line number information state machine has the following registers:

address	The program-counter value corresponding to a machine instruction generated by the compiler.
file	An unsigned integer indicating the identity of the source file corresponding to a machine instruction.
line	An unsigned integer indicating a source line number. Lines are numbered beginning at 1. The compiler may emit the value 0 in cases where an instruction cannot be attributed to any source line.

DWARF Debugging Information Format, Version 3

column	An unsigned integer indicating a column number within a source line. Columns are numbered beginning at 1. The value 0 is reserved to indicate that a statement begins at the “left edge” of the line.
is_stmt	A boolean indicating that the current instruction is a recommended breakpoint location. A recommended breakpoint location is intended to "represent" a line, a statement and/or a semantically distinct subpart of a statement.
basic_block	A boolean indicating that the current instruction is the beginning of a basic block.
end_sequence	A boolean indicating that the current address is that of the first byte after the end of a sequence of target machine instructions.
prologue_end	A boolean indicating that the current address is one (of possibly many) where execution should be suspended for an entry breakpoint of a function.
epilogue_begin	A boolean indicating that the current address is one (of possibly many) where execution should be suspended for an exit breakpoint of a function.
isa	An unsigned integer whose value encodes the applicable instruction set architecture for the current instruction.

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

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

address	0
file	1
line	1
column	0
is_stmt	determined by <code>default_is_stmt</code> in the line number program header
basic_block	"false"
end_sequence	"false"
prologue_end	"false"
epilogue_begin	"false"
isa	0

The 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, the object file description.

OTHER DEBUGGING INFORMATION

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).

DWARF Debugging Information Format, Version 3

4. `minimum_instruction_length` (ubyte)

The size in bytes of the smallest target machine instruction. Line number program opcodes that alter the address register first multiply their operands by this value.

5. `default_is_stmt` (ubyte)

The initial value of the `is_stmt` register.

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

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

6. `line_base` (sbyte)

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

7. `line_range` (ubyte)

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

8. `opcode_base` (ubyte)

The number assigned to the first special opcode.

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, 9 in DWARF 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.

OTHER DEBUGGING INFORMATION

9. `standard_opcode_lengths` (array of ubyte)

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

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

10. `include_directories` (sequence of path names)

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

11. `file_names` (sequence of file entries)

The sequence contains an entry for each source file that contributed to the line number information for this compilation unit or is used in other contexts, such as in a [declaration coordinate](#) or a macro file inclusion. Each entry consists of the following values:

- A null-terminated string containing the file name.
- An unsigned LEB128 number representing the directory index of the directory in which the file was found.
- An unsigned LEB128 number representing the (implementation-defined) time of last modification for the file.
- An unsigned LEB128 number representing the length in bytes of the file.

A compiler may choose to emit LEB128(0) for the time and length fields to indicate that this information is not available. The last entry is followed by a single null byte.

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

DWARF Debugging Information Format, Version 3

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

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

6.2.5 The Line Number Program

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

6.2.5.1 Special Opcodes

Each ubyte special opcode has the following effect on the state machine:

1. Add a signed integer to the `line` register.
2. Multiply an unsigned integer by the `minimum_instruction_length` field of the line number program header and add the result to the `address` register.
3. Append a row to the matrix using the current values of the state machine registers.
4. Set the `basic_block` register to “false.”
5. Set the `prologue_end` register to “false.”
6. Set the `epilogue_begin` register to “false.”

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

Instead of assigning a fixed meaning to each special opcode, the line number program uses several parameters in the header to configure the instruction set. There are two reasons for this. First, although the opcode space available for special opcodes now ranges from 13 through 255, the lower bound may increase if one adds new standard opcodes. Thus, the `opcode_base` field of the line number program header gives the value of the first special opcode. Second, the best choice of special-opcode meanings depends on the target architecture. For example, for a RISC machine where the compiler-generated code interleaves instructions from different lines to schedule the pipeline, it is important to be able to add a negative value to the `line` register to express the fact that a later instruction may have been emitted for an earlier source line. For a machine where pipeline scheduling never occurs, it is advantageous to trade away the ability to decrease the line register (a standard opcode provides an alternate way to decrease the line

OTHER DEBUGGING INFORMATION

number) in return for the ability to add larger positive values to the address register. To permit this variety of strategies, the line number program header defines a `line_base` field that specifies the minimum value which a special opcode can add to the line register and a `line_range` field that defines the range of values it can add to the line register.

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

$$\text{opcode} = (\text{desired line increment} - \text{line_base}) + (\text{line_range} * \text{address advance}) + \text{opcode_base}$$

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

To decode a special opcode, subtract the `opcode_base` from the opcode itself to give the adjusted opcode. The amount to increment the `address` register is the result of the adjusted opcode divided by the `line_range` multiplied by the `minimum_instruction_length` field from the header. That is,

$$\text{address increment} = (\text{adjusted opcode} / \text{line_range}) * \text{minimum_instruction_length}$$

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

$$\text{line increment} = \text{line_base} + (\text{adjusted opcode} \% \text{line_range})$$

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

DWARF Debugging Information Format, Version 3

The opcode mapping would be:

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

There is no requirement that the expression $255 - \text{line_base} + 1$ be an integral multiple of line_range .

6.2.5.2 Standard Opcodes

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

1. DW_LNS_copy

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

2. DW_LNS_advance_pc

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

OTHER DEBUGGING INFORMATION

3. DW_LNS_advance_line

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

4. DW_LNS_set_file

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

5. DW_LNS_set_column

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

6. DW_LNS_negate_stmt

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

7. DW_LNS_set_basic_block

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

8. DW_LNS_const_add_pc

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

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

9. DW_LNS_fixed_advance_pc

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

DWARF Debugging Information Format, Version 3

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.

OTHER DEBUGGING INFORMATION

12. DW_LNS_set_isa

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

6.2.5.3 Extended Opcodes

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

1. DW_LNE_end_sequence

The DW_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 appropriate to hold an address on the target machine. It sets the `address` register to the value given by the relocatable address.

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

3. DW_LNE_define_file

The DW_LNE_define_file opcode takes four operands:

1. A null-terminated string containing a source file name.
2. An unsigned LEB128 number representing the directory index of the directory in which the file was found.
3. An unsigned LEB128 number representing the time of last modification of the file.
4. An unsigned LEB128 number representing the length in bytes of the file.

The time and length fields may contain LEB128(0) if the information is not available.

DWARF Debugging Information Format, Version 3

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

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

Appendix D.4 gives some sample line number programs.

6.3 Macro Information

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

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

6.3.1 Macinfo Types

The valid macinfo types are as follows:

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

OTHER DEBUGGING INFORMATION

6.3.1.1 Define and Undefine Entries

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

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

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

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

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

6.3.1.2 Start File Entries

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

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

6.3.1.3 End File Entries

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

DWARF Debugging Information Format, Version 3

6.3.1.4 Vendor Extension Entries

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

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

6.3.2 Base Source Entries

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

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

6.3.3 Macinfo Entries for Command Line Options

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

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

6.3.4 General Rules and Restrictions

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

OTHER DEBUGGING INFORMATION

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

6.4 Call Frame Information

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

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

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

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

DWARF Debugging Information Format, Version 3

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

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

6.4.1 Structure of Call Frame Information

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

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

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

OTHER DEBUGGING INFORMATION

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

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

The register rules are:

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

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

The virtual unwind information is encoded in a self-contained section called `.debug_frame`. Entries in a `.debug_frame` section are aligned on an addressing unit boundary and come in two forms: A Common Information Entry (CIE) and a Frame Description Entry (FDE).

DWARF Debugging Information Format, Version 3

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

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

1. `length` (initial length)

A constant that gives the number of bytes of the CIE structure, not including the `length` field itself (where `length mod <size of an address> == 0`) (see Section 7.2.2).

2. `CIE_id` (see Section 7.4)

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

3. `version` (ubyte)

A version number (see Appendix F). 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.

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

5. `code_alignment_factor` (unsigned LEB128)

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

6. `data_alignment_factor` (signed LEB128)

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

OTHER DEBUGGING INFORMATION

7. `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.

In DWARF Version 2, this field is a ubyte.

8. `initial_instructions` (array of ubyte)

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

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.

In DWARF Version 2, the default value was not mentioned.

9. `padding` (array of ubyte)

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

An FDE contains the following fields, in order:

1. `length` (initial length)

A constant that gives the number of bytes of the header and instruction stream for this function, not including the `length` field itself (where `length mod addressing_unit_size == 0`) (see Section 7.2.2).

2. `CIE_pointer` (section offset)

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

3. `initial_location` (target address)

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

DWARF Debugging Information Format, Version 3

4. `address_range` (target address)

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

5. `instructions` (array of ubyte)

A sequence of table defining instructions that are described below.

6. `padding` (array of ubyte)

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

6.4.2 Call Frame Instructions

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

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.

6.4.2.1 Row Creation Instructions

1. `DW_CFA_set_loc`

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

OTHER DEBUGGING INFORMATION

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 $\text{delta} * \text{code_alignment_factor}$. 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.

DWARF Debugging Information Format, Version 3

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).

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).

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.

6. DW_CFA_def_cfa_expression

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

The DW_OP_call2, DW_OP_call4, DW_OP_call_ref, DW_OP_push_object_address and DW_OP_call_frame_cfa DWARF operators (see Section 2.5.1) cannot be used in such a DWARF expression.

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.”

OTHER DEBUGGING INFORMATION

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 with a value of ($N = \text{factored offset} * \text{data_alignment_factor}$).

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.

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 with a value of ($N = \text{factored offset} * \text{data_alignment_factor}$).

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.

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 the second register.

DWARF Debugging Information Format, Version 3

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.

The DW_OP_call2, DW_OP_call4, DW_OP_call_ref, DW_OP_push_object_address and DW_OP_call_frame_cfa DWARF operators (see Section 2.5.1) cannot be used in such a DWARF expression.

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.

The DW_OP_call2, DW_OP_call4, DW_OP_call_ref and DW_OP_push_object_address DWARF operators (see Section 2.5.1) cannot be used in such a DWARF expression.

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.

OTHER DEBUGGING INFORMATION

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 was encountered, then compute a new location value (L2). If $L1 \geq L2$ then process the instruction and go back to step 2.*
- 4. The end of the instruction stream can be thought of as a `DW_CFA_set_loc` (`initial_location` + `address_range`) instruction. Unless the FDE is ill-formed, L1 should be less than L2 at this point.*

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

For an example, see Appendix D.6.

DWARF Debugging Information Format, Version 3

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.

DWARF Debugging Information Format, Version 3

7.2 Reserved Values

7.2.1 Error Values

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

7.2.2 Initial Length Values

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

In an initial length field, the values `0xffffffff0` through `0xfffffffff` 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, `0xfffffffff`, is defined below (see Section 7.4); the use of the other values is reserved for possible future extensions.

7.3 Executable Objects and Shared Objects

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

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

7.4 32-Bit and 64-Bit DWARF Formats

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

DWARF Version 3 defines two closely related file formats. In the 32-bit DWARF format, all values that represent lengths of DWARF sections and offsets relative to the beginning of

DATA REPRESENTATION

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

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

1. In the 32-bit DWARF format, an **initial length field** (see Section 7.2.2) is an unsigned 32-bit integer (which must be less than `0xffffffff00`); 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.

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

The `CIE_id` field in a CIE structure must be 64 bits because it overlays the `CIE_pointer` in a FDE structure; this implicit union must be accessed to distinguish whether a CIE or

DWARF Debugging Information Format, Version 3

FDE is present, consequently, these two fields must exactly overlay each other (both offset and size).

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

<u>Form</u>	<u>Role</u>
DW_FORM_ref_addr	offset in <code>.debug_info</code>
DW_FORM_strp	offset in <code>.debug_str</code>
DW_OP_call_ref	offset in <code>.debug_info</code>

4. Within the body of the `.debug_info` section, certain classes of attribute value use an attribute value form that depends on the DWARF format as follows: in the 32-bit DWARF format, the `lineptr`, `loclistptr`, `macptr` and `rangelistptr` classes use form `DW_FORM_data4`; in the 64-bit DWARF format, they use form `DW_FORM_data8`.
5. Within the body of the `.debug_pubnames` and `.debug_pubtypes` sections, the representation of the first field of each tuple (which represents an offset in the `.debug_info` section) depends on the DWARF format as follows: in the 32-bit DWARF format, this field is a 32-bit unsigned integer; in the 64-bit DWARF format, it is a 64-bit unsigned integer.

The 32-bit and 64-bit DWARF format conventions must 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 be compiled using the 64-bit format; separate 32-bit format versions of system supplied shared executable libraries can still be used.

7.5 Format of Debugging Information

For each compilation unit compiled with a DWARF Version 3 producer, a contribution is made to the `.debug_info` section of the object file. Each such contribution consists of a compilation unit header (see Section 7.5.1) followed by a single `DW_TAG_compile_unit` or `DW_TAG_partial_unit` debugging information entry, together with its children.

Each debugging information entry begins with a code that represents an entry in a separate abbreviations table. This code is followed directly by a series of attribute values. The appropriate entry in the abbreviations table guides the interpretation of the information contained directly in the `.debug_info` section. Each compilation unit is associated with a particular abbreviation table, but multiple compilation units may share the same table.

7.5.1 Compilation Unit Header

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

1. `unit_length` (initial length)

A 4-byte or 12-byte unsigned integer representing the length of the `.debug_info` contribution for that compilation unit, not including the length field itself. In the [32-bit DWARF format](#), this is a 4-byte unsigned integer (which must be less than `0xffffffff00`); 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).

4. `version` (uhalf)

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

5. `debug_abbrev_offset` (section offset)

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

6. `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.

DWARF Debugging Information Format, Version 3

7.5.2 Debugging Information Entry

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

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

7.5.3 Abbreviations Tables

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

The abbreviations table for a single compilation unit consists of a series of abbreviation declarations. Each declaration specifies the tag and attributes for a particular form of debugging information entry. Each declaration begins with an unsigned LEB128 number representing the abbreviation code itself. It is this code that appears at the beginning of a debugging information entry in the `.debug_info` section. As described above, the abbreviation code 0 is reserved for null debugging information entries. The abbreviation code is followed by another unsigned LEB128 number that encodes the entry's tag. The encodings for the tag names are given in [Figure 18](#).

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](#). (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.

DATA REPRESENTATION

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

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

See [Appendix D.1](#) for a depiction of the organization of the debugging information.

7.5.4 Attribute Encodings

The encodings for the attribute names are given in [Figure 20](#).

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

Some encodings are members of more than one class; in that case, the list of classes allowed by the applicable attribute in [Figure 20](#) determines the class of the form. `DW_FORM_data4` and `DW_FORM_data8` may be members of classes `constant`, `lineptr`, `loclistptr`, `macptr` and `rangelistptr`. They are members of the class `constant` if used for the value of an attribute that allows class `constant` but not class `lineptr`, `loclistptr`, `macptr` or `rangelistptr`. They are members of the class `lineptr`, `loclistptr`, `macptr` or `rangelistptr` if used for the value of an attribute that allows one of those classes.

In DWARF Version 2, each form belonged to exactly one class so that the classes formed a partition of the set of forms. Moreover, the class `constant` was used for some “pointers” (described as offsets) into other sections. In practice such offsets had to use `DW_FORM_data4` since they had to be a form that could be relocated during linking (making such “constants” not very constant).

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

DWARF Debugging Information Format, Version 3

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

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

address

Represented as an object of appropriate size to hold an address on the target machine (DW_FORM_addr). The size is encoded in the compilation unit header (see Section 7.5.1). This address is relocatable in a relocatable object file and is relocated in an executable file or shared object.

block

Blocks come in four forms:

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

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

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

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

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

constant

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

DATA REPRESENTATION

in question does not allow one of the classes `lineptr`, `loclistptr`, `macptr` or `rangelistptr` (see below).

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 an offset to a debugging information entry, 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>`.

flag

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

lineptr

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

loclistptr

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

DWARF Debugging Information Format, Version 3

macptr

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

rangelistptr

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

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

reference

There are two types of reference.

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

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

DATA REPRESENTATION

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

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

The use of compilation unit relative references will reduce the number of link-time relocations and so speed up linking. The use of the second type of references allows for the commonization of information, such as types, across compilation units.

string

A string is a sequence of contiguous non-null bytes followed by one null byte. A string may be represented immediately in the debugging information entry itself (`DW_FORM_string`), or may be represented as an offset into a string table contained in the `.debug_str` section of the object file (`DW_FORM_strp`). In the [32-bit DWARF format](#), the representation of a `DW_FORM_strp` value is an 4-byte unsigned offset; in the [64-bit DWARF format](#), it is an 8-byte unsigned offset (see [Section 7.4](#)).

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

The Unicode Standard 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.

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

The form encodings are listed in [Figure 21](#).

DWARF Debugging Information Format, Version 3

Tag name	Value
DW_TAG_array_type	0x01
DW_TAG_class_type	0x02
DW_TAG_entry_point	0x03
DW_TAG_enumeration_type	0x04
DW_TAG_formal_parameter	0x05
DW_TAG_imported_declaration	0x08
DW_TAG_label	0x0a
DW_TAG_lexical_block	0x0b
DW_TAG_member	0x0d
DW_TAG_pointer_type	0x0f
DW_TAG_reference_type	0x10
DW_TAG_compile_unit	0x11
DW_TAG_string_type	0x12
DW_TAG_structure_type	0x13
DW_TAG_subroutine_type	0x15
DW_TAG_typedef	0x16
DW_TAG_union_type	0x17
DW_TAG_unspecified_parameters	0x18
DW_TAG_variant	0x19
DW_TAG_common_block	0x1a

DATA REPRESENTATION

Tag name	Value
DW_TAG_common_inclusion	0x1b
DW_TAG_inheritance	0x1c
DW_TAG_inlined_subroutine	0x1d
DW_TAG_module	0x1e
DW_TAG_ptr_to_member_type	0x1f
DW_TAG_set_type	0x20
DW_TAG_subrange_type	0x21
DW_TAG_with_stmt	0x22
DW_TAG_access_declaration	0x23
DW_TAG_base_type	0x24
DW_TAG_catch_block	0x25
DW_TAG_const_type	0x26
DW_TAG_constant	0x27
DW_TAG_enumerator	0x28
DW_TAG_file_type	0x29
DW_TAG_friend	0x2a
DW_TAG_namelist	0x2b
DW_TAG_namelist_item	0x2c
DW_TAG_packed_type	0x2d
DW_TAG_subprogram	0x2e

DWARF Debugging Information Format, Version 3

Tag name	Value
DW_TAG_template_type_parameter	0x2f
DW_TAG_template_value_parameter	0x30
DW_TAG_thrown_type	0x31
DW_TAG_try_block	0x32
DW_TAG_variant_part	0x33
DW_TAG_variable	0x34
DW_TAG_volatile_type	0x35
DW_TAG_dwarf_procedure †	0x36
DW_TAG_restrict_type †	0x37
DW_TAG_interface_type †	0x38
DW_TAG_namespace †	0x39
DW_TAG_imported_module †	0x3a
DW_TAG_unspecified_type †	0x3b
DW_TAG_partial_unit †	0x3c
DW_TAG_imported_unit †	0x3d
DW_TAG_condition †	0x3f

DATA REPRESENTATION

Tag name	Value
DW_TAG_shared_type ‡	0x40
DW_TAG_lo_user	0x4080
DW_TAG_hi_user	0xffff

‡ New in DWARF Version 3

Figure 18. Tag encodings

Child determination name	Value
DW_CHILDREN_no	0x00
DW_CHILDREN_yes	0x01

Figure 19. Child determination encodings

Attribute name	Value	Classes
DW_AT_sibling	0x01	reference
DW_AT_location	0x02	block, loclistptr
DW_AT_name	0x03	string
DW_AT_ordering	0x09	constant
DW_AT_byte_size	0x0b	block, constant, reference
DW_AT_bit_offset	0x0c	block, constant, reference
DW_AT_bit_size	0x0d	block, constant, reference

DWARF Debugging Information Format, Version 3

Attribute name	Value	Classes
DW_AT_stmt_list	0x10	lineptr
DW_AT_low_pc	0x11	address
DW_AT_high_pc	0x12	address
DW_AT_language	0x13	constant
DW_AT_discr	0x15	reference
DW_AT_discr_value	0x16	constant
DW_AT_visibility	0x17	constant
DW_AT_import	0x18	reference
DW_AT_string_length	0x19	block, loclistptr
DW_AT_common_reference	0x1a	reference
DW_AT_comp_dir	0x1b	string
DW_AT_const_value	0x1c	block, constant, string
DW_AT_containing_type	0x1d	reference
DW_AT_default_value	0x1e	reference
DW_AT_inline	0x20	constant
DW_AT_is_optional	0x21	flag
DW_AT_lower_bound	0x22	block, constant, reference
DW_AT_producer	0x25	string
DW_AT_prototyped	0x27	flag
DW_AT_return_addr	0x2a	block, loclistptr

DATA REPRESENTATION

Attribute name	Value	Classes
DW_AT_start_scope	0x2c	constant
DW_AT_bit_stride	0x2e	constant
DW_AT_upper_bound	0x2f	block, constant, reference
DW_AT_abstract_origin	0x31	reference
DW_AT_accessibility	0x32	constant
DW_AT_address_class	0x33	constant
DW_AT_artificial	0x34	flag
DW_AT_base_types	0x35	reference
DW_AT_calling_convention	0x36	constant
DW_AT_count	0x37	block, constant, reference
DW_AT_data_member_location	0x38	block, constant, loclistptr
DW_AT_decl_column	0x39	constant
DW_AT_decl_file	0x3a	constant
DW_AT_decl_line	0x3b	constant
DW_AT_declaration	0x3c	flag
DW_AT_discr_list	0x3d	block
DW_AT_encoding	0x3e	constant
DW_AT_external	0x3f	flag
DW_AT_frame_base	0x40	block, loclistptr
DW_AT_friend	0x41	reference

DWARF Debugging Information Format, Version 3

Attribute name	Value	Classes
DW_AT_identifier_case	0x42	constant
DW_AT_macro_info	0x43	macptr
DW_AT_namelist_item	0x44	block
DW_AT_priority	0x45	reference
DW_AT_segment	0x46	block, loclistptr
DW_AT_specification	0x47	reference
DW_AT_static_link	0x48	block, loclistptr
DW_AT_type	0x49	reference
DW_AT_use_location	0x4a	block, loclistptr
DW_AT_variable_parameter	0x4b	flag
DW_AT_virtuality	0x4c	constant
DW_AT_vtable_elem_location	0x4d	block, loclistptr
DW_AT_allocated †	0x4e	block, constant, reference
DW_AT_associated †	0x4f	block, constant, reference
DW_AT_data_location †	0x50	block
DW_AT_byte_stride †	0x51	block, constant, reference
DW_AT_entry_pc †	0x52	address
DW_AT_use_UTF8 †	0x53	flag
DW_AT_extension †	0x54	reference
DW_AT_ranges †	0x55	rangelistptr

DATA REPRESENTATION

Attribute name	Value	Classes
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
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

DWARF Debugging Information Format, Version 3

Attribute name	Value	Classes
DW_AT_recursive ‡	0x68	flag
DW_AT_lo_user	0x2000	---
DW_AT_hi_user	0x3fff	---

‡ New in DWARF Version 3

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, lineptr, loclistptr, macptr, rangelistptr
DW_FORM_data8	0x07	constant, lineptr, loclistptr, macptr, rangelistptr
DW_FORM_string	0x08	string
DW_FORM_block	0x09	block
DW_FORM_block1	0x0a	block
DW_FORM_data1	0x0b	constant
DW_FORM_flag	0x0c	flag
DW_FORM_sdata	0x0d	constant

DATA REPRESENTATION

Form name	Value	Class
DW_FORM_strp	0x0e	string
DW_FORM_udata	0x0f	constant
DW_FORM_ref_addr	0x10	reference
DW_FORM_ref1	0x11	reference
DW_FORM_ref2	0x12	reference
DW_FORM_ref4	0x13	reference
DW_FORM_ref8	0x14	reference
DW_FORM_ref_udata	0x15	reference
DW_FORM_indirect	0x16	(see Section 7.5.3)

Figure 21. Attribute form encodings

7.6 Variable Length Data

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

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

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

DWARF Debugging Information Format, Version 3

Figure 22 gives some examples of `DW_FORM_adata` numbers. The 0x80 in each case is the high order bit of the byte, indicating that an additional byte follows.

The encoding for `DW_FORM_sdata` (signed, two's complement LEB128) numbers is similar, except that the criterion for discarding high order bytes is not whether they are zero, but whether they consist entirely of sign extension bits. Consider the 32-bit integer -2 . The three high level bytes of the number are sign extension, thus LEB128 would represent it as a single byte containing the low order 7 bits, with the high order bit cleared to indicate the end of the byte stream. Note that there is nothing within the LEB128 representation that indicates whether an encoded number is signed or unsigned. The decoder must know what type of number to expect.

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

Figure 22. Examples of unsigned LEB128 encodings

DATA REPRESENTATION

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

Figure 23 gives some examples of [DW_FORM_sdata](#) numbers.

[Appendix C](#) gives algorithms for encoding and decoding these forms.

7.7 DWARF Expressions and Location Descriptions

7.7.1 DWARF Expressions

A DWARF expression is stored in a block of contiguous bytes. The bytes form a set of operations. Each location operation has a 1-byte code that identifies that operation. Operations can be followed by one or more bytes of additional data. All operations in a DWARF expression are concatenated from left to right. The encodings for the operations in a DWARF expression are described in [Figure 24](#).

DWARF Debugging Information Format, Version 3

Operation	Code	No. of Operands	Notes
DW_OP_addr	0x03	1	constant address (size target specific)
DW_OP_deref	0x06	0	
DW_OP_const1u	0x08	1	1-byte constant
DW_OP_const1s	0x09	1	1-byte constant
DW_OP_const2u	0x0a	1	2-byte constant
DW_OP_const2s	0x0b	1	2-byte constant
DW_OP_const4u	0x0c	1	4-byte constant
DW_OP_const4s	0x0d	1	4-byte constant
DW_OP_const8u	0x0e	1	8-byte constant
DW_OP_const8s	0x0f	1	8-byte constant
DW_OP_constu	0x10	1	ULEB128 constant
DW_OP_consts	0x11	1	SLEB128 constant
DW_OP_dup	0x12	0	
DW_OP_drop	0x13	0	
DW_OP_over	0x14	0	
DW_OP_pick	0x15	1	1-byte stack index
DW_OP_swap	0x16	0	
DW_OP_rot	0x17	0	

DATA REPRESENTATION

Operation	Code	No. of Operands	Notes
DW_OP_xderef	0x18	0	
DW_OP_abs	0x19	0	
DW_OP_and	0x1a	0	
DW_OP_div	0x1b	0	
DW_OP_minus	0x1c	0	
DW_OP_mod	0x1d	0	
DW_OP_mul	0x1e	0	
DW_OP_neg	0x1f	0	
DW_OP_not	0x20	0	
DW_OP_or	0x21	0	
DW_OP_plus	0x22	0	
DW_OP_plus_uconst	0x23	1	ULEB128 addend
DW_OP_shl	0x24	0	
DW_OP_shr	0x25	0	
DW_OP_shra	0x26	0	
DW_OP_xor	0x27	0	
DW_OP_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	

DWARF Debugging Information Format, Version 3

Operation	Code	No. of Operands	Notes
DW_OP_gt	0x2b	0	
DW_OP_le	0x2c	0	
DW_OP_lt	0x2d	0	
DW_OP_ne	0x2e	0	
DW_OP_lit0	0x30	0	literals 0..31 = (DW_OP_lit0 + literal)
DW_OP_lit1	0x31	0	
...			
DW_OP_lit31	0x4f	0	
DW_OP_reg0	0x50	0	reg 0..31 = (DW_OP_reg0 + regnum)
DW_OP_reg1	0x51	0	
...			
DW_OP_reg31	0x6f	0	
DW_OP_breg0	0x70	1	SLEB128 offset 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

DATA REPRESENTATION

Operation	Code	No. of Operands	Notes
DW_OP_piece	0x93	1	ULEB128 size of piece addressed
DW_OP_deref_size	0x94	1	1-byte size of data retrieved
DW_OP_xderef_size	0x95	1	1-byte size of data retrieved
DW_OP_nop	0x96	0	
DW_OP_push_object_address ‡	0x97	0	
DW_OP_call2 ‡	0x98	1	2-byte offset of DIE
DW_OP_call4 ‡	0x99	1	4-byte offset of DIE
DW_OP_call_ref ‡	0x9a	1	4- or 8-byte offset of DIE
DW_OP_form_tls_address ‡	0x9b	0	
DW_OP_call_frame_cfa ‡	0x9c	0	
DW_OP_bit_piece ‡	0x9d	2	
DW_OP_lo_user	0xe0		
DW_OP_hi_user	0xff		

‡ New in DWARF Version 3

Figure 24. DWARF operation encodings

7.7.2 Location Expressions

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

7.7.3 Location Lists

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

DWARF Debugging Information Format, Version 3

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

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

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

7.8 Base Type Encodings

The values of the constants used in the [DW_AT_encoding](#) attribute are given in [Figure 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

DATA REPRESENTATION

Base type encoding code name	Value
DW_ATE_unsigned_fixed ‡	0x0e
DW_ATE_decimal_float ‡	0x0f
DW_ATE_lo_user	0x80
DW_ATE_hi_user	0xff

‡ New in DWARF Version 3

Figure 25. Base type encoding values

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

‡ New in DWARF Version 3

Figure 26. Decimal sign encodings

DWARF Debugging Information Format, Version 3

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

‡ New in DWARF Version 3

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

DATA REPRESENTATION

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
<code>DW_VIS_local</code>	0x01
<code>DW_VIS_exported</code>	0x02
<code>DW_VIS_qualified</code>	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
<code>DW_VIRTUALITY_none</code>	0x00
<code>DW_VIRTUALITY_virtual</code>	0x01
<code>DW_VIRTUALITY_pure_virtual</code>	0x02

Figure 30. Virtuality encodings

7.12 Source Languages

The encodings for source languages are given in [Figure 31](#). Names marked with † and their associated values are reserved, but the languages they represent are not well supported in DWARF Version 3.

DWARF Debugging Information Format, Version 3

Language name	Value
DW_LANG_C89	0x0001
DW_LANG_C	0x0002
DW_LANG_Ada83 [†]	0x0003
DW_LANG_C_plus_plus	0x0004
DW_LANG_Cobol74 [†]	0x0005
DW_LANG_Cobol85 [†]	0x0006
DW_LANG_Fortran77	0x0007
DW_LANG_Fortran90	0x0008
DW_LANG_Pascal83	0x0009
DW_LANG_Modula2	0x000a
DW_LANG_Java [‡]	0x000b
DW_LANG_C99 [‡]	0x000c
DW_LANG_Ada95 ^{†‡}	0x000d
DW_LANG_Fortran95 [‡]	0x000e
DW_LANG_PLI ^{†‡}	0x000f

DATA REPRESENTATION

Language name	Value
DW_LANG_ObjC †	0x0010
DW_LANG_ObjC_plus_plus †	0x0011
DW_LANG_UPC †	0x0012
DW_LANG_D †	0x0013
DW_LANG_lo_user	0x8000
DW_LANG_hi_user	0xffff

† See text ‡ New in DWARF Version 3

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

DWARF Debugging Information Format, Version 3

7.15 Calling Convention Encodings

The encodings for the values of the `DW_AT_calling_convention` attribute are given in [Figure 33](#).

Calling Convention Name	Value
<code>DW_CC_normal</code>	0x01
<code>DW_CC_program</code>	0x02
<code>DW_CC_nocall</code>	0x03
<code>DW_CC_lo_user</code>	0x40
<code>DW_CC_hi_user</code>	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
<code>DW_INL_not_inlined</code>	0x00
<code>DW_INL_inlined</code>	0x01
<code>DW_INL_declared_not_inlined</code>	0x02
<code>DW_INL_declared_inlined</code>	0x03

Figure 34. Inline encodings

DATA REPRESENTATION

7.17 Array Ordering

The encodings for the values of the ordering attribute of arrays used in the `DW_AT_ordering` attribute are given in [Figure 35](#).

Ordering name	Value
<code>DW_ORD_row_major</code>	0x00
<code>DW_ORD_col_major</code>	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
<code>DW_DSC_label</code>	0x00
<code>DW_DSC_range</code>	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 `0xffffffff00`); 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](#)).

DWARF Debugging Information Format, Version 3

2. `version` (uhalf)

A 2-byte version identifier containing the value 2 for DWARF Version 3 (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 `0xffffffff00`); 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 for DWARF Version 3 (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).

DATA REPRESENTATION

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 descriptor on the target system.

This header is followed by a series of tuples. Each tuple consists of an address and a length, each in the size appropriate for an address on the target architecture. The first tuple following the header in each set begins at an offset that is a multiple of the size of a single tuple (that is, twice the size of an address). The header is padded, if necessary, to the appropriate boundary. Each set of tuples is terminated by a 0 for the address and 0 for the length.

7.21 Line Number Information

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

The boolean values “true” and “false” used by the line number information program are encoded as a single byte containing the value 0 for “false,” and a non-zero value for “true.”

The encodings for the pre-defined standard opcodes are given in [Figure 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

DWARF Debugging Information Format, Version 3

Opcode Name	Value
DW_LNS_fixed_advance_pc	0x09
DW_LNS_set_prologue_end ‡	0x0a
DW_LNS_set_epilogue_begin ‡	0x0b
DW_LNS_set_isa ‡	0x0c

‡ New in DWARF Version 3

Figure 37. Line Number Standard Opcode Encodings

The encodings for the predefined 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_lo_user ‡	0x80
DW_LNE_hi_user ‡	0xff

‡ New in DWARF Version 3

Figure 38. Line Number Extended Opcode Encodings

DATA REPRESENTATION

7.22 Macro Information

The source line numbers and source file indices encoded in the macro information section are represented as unsigned LEB128 numbers as are the constants in an [DW_MACROINFO_vendor_ext](#) entry.

The macinfo type is encoded as a single byte. The encodings are given in [Figure 39](#).

Macinfo Type Name	Value
DW_MACROINFO_define	0x01
DW_MACROINFO_undef	0x02
DW_MACROINFO_start_file	0x03
DW_MACROINFO_end_file	0x04
DW_MACROINFO_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 3 (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](#).

DWARF Debugging Information Format, Version 3

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

DATA REPRESENTATION

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		

‡ New in DWARF Version 3

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 used by [DW_FORM_addr](#) on the target machine.

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

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

DWARF Debugging Information Format, Version 3

7.25 Dependencies and Constraints

The debugging information in this format is intended to exist in the `.debug_abbrev`, `.debug_aranges`, `.debug_frame`, `.debug_info`, `.debug_line`, `.debug_loc`, `.debug_macinfo`, `.debug_pubnames`, `.debug_pubtypes`, `.debug_ranges` and `.debug_str` sections of an object file, 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 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 an 8-byte address or 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.

DATA REPRESENTATION

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

DWARF Debugging Information Format, Version 3

DWARF Debugging Information Format, Version 3

Appendix A -- Attributes by Tag Value (informative)

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

In the following table, DECL means [DW_AT_decl_column](#), [DW_AT_decl_file](#), and [DW_AT_decl_line](#).

TAG Name	Applicable Attributes
DW_TAG_access_declaration	DECL DW_AT_accessibility DW_AT_description DW_AT_name DW_AT_sibling
DW_TAG_array_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_allocated DW_AT_associated DW_AT_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

DWARF Debugging Information Format, Version 3

DW_TAG_base_type	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_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	DW_AT_abstract_origin DW_AT_high_pc DW_AT_low_pc DW_AT_ranges DW_AT_segment DW_AT_sibling
DW_TAG_class_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_allocated DW_AT_associated DW_AT_byte_size DW_AT_data_location DW_AT_declaration DW_AT_description DW_AT_name DW_AT_sibling DW_AT_specification DW_AT_start_scope DW_AT_visibility

Appendix -- Attributes by Tag Value (informative)

DW_TAG_common_block	DECL DW_AT_declaration DW_AT_description DW_AT_location DW_AT_name DW_AT_segment DW_AT_sibling DW_AT_visibility
DW_TAG_common_inclusion	DECL DW_AT_common_reference DW_AT_declaration DW_AT_sibling DW_AT_visibility
DW_TAG_compile_unit	DW_AT_base_types DW_AT_comp_dir DW_AT_identifier_case DW_AT_high_pc DW_AT_language DW_AT_low_pc DW_AT_macro_info DW_AT_name DW_AT_producer DW_AT_ranges DW_AT_segment DW_AT_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

DWARF Debugging Information Format, Version 3

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_name DW_AT_sibling DW_AT_start_scope DW_AT_type DW_AT_visibility
DW_TAG_dwarf_procedure	DW_AT_location
DW_TAG_entry_point	DW_AT_address_class DW_AT_description DW_AT_frame_base DW_AT_low_pc DW_AT_name DW_AT_return_addr DW_AT_segment DW_AT_sibling DW_AT_static_link DW_AT_type

Appendix -- Attributes by Tag Value (informative)

<p>DW_TAG_enumeration_type</p>	<p>DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_allocated DW_AT_associated DW_AT_bit_stride DW_AT_byte_size DW_AT_byte_stride DW_AT_data_location DW_AT_declaration DW_AT_description DW_AT_name DW_AT_sibling DW_AT_specification DW_AT_start_scope DW_AT_type DW_AT_visibility</p>
<p>DW_TAG_enumerator</p>	<p>DECL DW_AT_const_value DW_AT_description DW_AT_name DW_AT_sibling</p>
<p>DW_TAG_file_type</p>	<p>DECL DW_AT_abstract_origin DW_AT_allocated DW_AT_associated DW_AT_byte_size DW_AT_data_location DW_AT_description DW_AT_name DW_AT_sibling DW_AT_start_scope DW_AT_type DW_AT_visibility</p>

DWARF Debugging Information Format, Version 3

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

Appendix -- Attributes by Tag Value (informative)

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_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

DWARF Debugging Information Format, Version 3

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

Appendix -- Attributes by Tag Value (informative)

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

DWARF Debugging Information Format, Version 3

DW_TAG_partial_unit	DW_AT_base_types DW_AT_comp_dir DW_AT_description DW_AT_identifier_case DW_AT_high_pc DW_AT_language DW_AT_low_pc DW_AT_macro_info DW_AT_name DW_AT_producer DW_AT_ranges DW_AT_segment DW_AT_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_specification DW_AT_type
DW_TAG_ptr_to_member_type	DECL DW_AT_abstract_origin DW_AT_address_class DW_AT_allocated DW_AT_associated DW_AT_containing_type DW_AT_data_location DW_AT_declaration DW_AT_description DW_AT_name DW_AT_sibling DW_AT_type DW_AT_use_location DW_AT_visibility

Appendix -- Attributes by Tag Value (informative)

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_set_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_allocated DW_AT_associated DW_AT_byte_size DW_AT_data_location DW_AT_declaration DW_AT_description DW_AT_name DW_AT_start_scope DW_AT_sibling DW_AT_type DW_AT_visibility
DW_TAG_shared_type	DECL DW_AT_count DW_AT_name DW_AT_sibling DW_AT_type

DWARF Debugging Information Format, Version 3

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

Appendix -- Attributes by Tag Value (informative)

<p>DW_TAG_subprogram</p>	<p>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_low_pc 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</p>
--------------------------	---

DWARF Debugging Information Format, Version 3

DW_TAG_subrange_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_allocated DW_AT_associated 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

Appendix -- Attributes by Tag Value (informative)

DW_TAG_template_type_parameter	DECL DW_AT_description DW_AT_name DW_AT_sibling DW_AT_type
DW_TAG_template_value_parameter	DECL DW_AT_const_value DW_AT_description DW_AT_name DW_AT_sibling DW_AT_type
DW_TAG_thrown_type	DECL DW_AT_allocated DW_AT_associated DW_AT_data_location DW_AT_sibling DW_AT_type
DW_TAG_try_block	DW_AT_abstract_origin DW_AT_high_pc DW_AT_low_pc DW_AT_ranges DW_AT_segment DW_AT_sibling
DW_TAG_typedef	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_allocated DW_AT_associated DW_AT_data_location DW_AT_declaration DW_AT_description DW_AT_name DW_AT_sibling DW_AT_start_scope DW_AT_type DW_AT_visibility

DWARF Debugging Information Format, Version 3

DW_TAG_union_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_allocated DW_AT_associated DW_AT_byte_size DW_AT_data_location DW_AT_declaration DW_AT_description DW_AT_name DW_AT_sibling DW_AT_specification DW_AT_start_scope DW_AT_visibility
DW_TAG_unspecified_parameters	DECL DW_AT_abstract_origin DW_AT_artificial DW_AT_sibling
DW_TAG_unspecified_type	DECL DW_AT_description DW_AT_name
DW_TAG_variable	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_const_value DW_AT_declaration DW_AT_description DW_AT_endianity DW_AT_external DW_AT_location DW_AT_name DW_AT_segment DW_AT_sibling DW_AT_specification DW_AT_start_scope DW_AT_type DW_AT_visibility

Appendix -- Attributes by Tag Value (informative)

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	DECL 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

DWARF Debugging Information Format, Version 3

DWARF Debugging Information Format, Version 3

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.

DWARF Debugging Information Format, Version 3

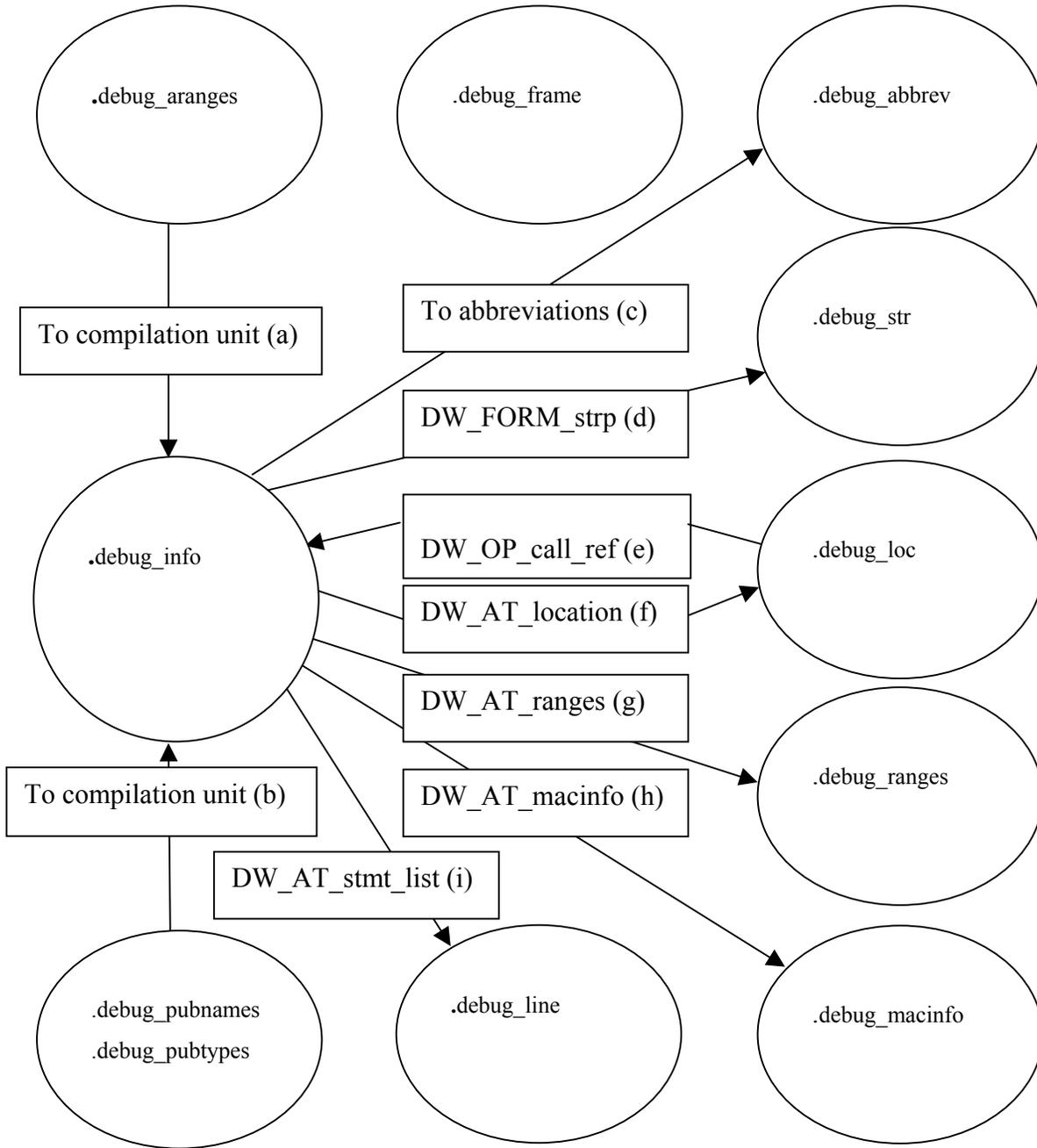


Figure 43. Debug section relationships

Appendix -- Debug Section Relationships (informative)

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` The `debug_abbrev_offset` value in the header is the offset in the `.debug_abbrev` section of the abbreviations for that compilation unit.
- (d) `.debug_info` Attribute values of class string may have form `DW_FORM_strp`, whose value is the offset in the `.debug_str` section of the corresponding string.
- (e) `.debug_loc` The operand of the `DW_OP_call_ref` DWARF expression operator is the offset of a debugging information entry in the `.debug_info` section.
- (f) `.debug_info` An attribute value of class `loclistptr` (specifically form `DW_FORM_data4/8`) is an offset within the `.debug_loc` section of a location list.
- (g) `.debug_info` An attribute value of class `rangelistptr` (specifically form `DW_FORM_data4/8`) is an offset within the `.debug_ranges` section of a range list.
- (h) `.debug_info` An attribute value of class `macptr` (specifically form `DW_FORM_data4/8`) is an offset within the `.debug_macro` section of the beginning of the macro information for the referencing unit.
- (i) `.debug_info` An attribute value of class `lineptr` (specifically form `DW_FORM_data4/8`) is an offset in the `.debug_line` section of the beginning of the line number information for the referencing unit.

DWARF Debugging Information Format, Version 3

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

DWARF Debugging Information Format, Version 3

```
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

DWARF Debugging Information Format, Version 3

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;
-----
```

DWARF Debugging Information Format, Version 3

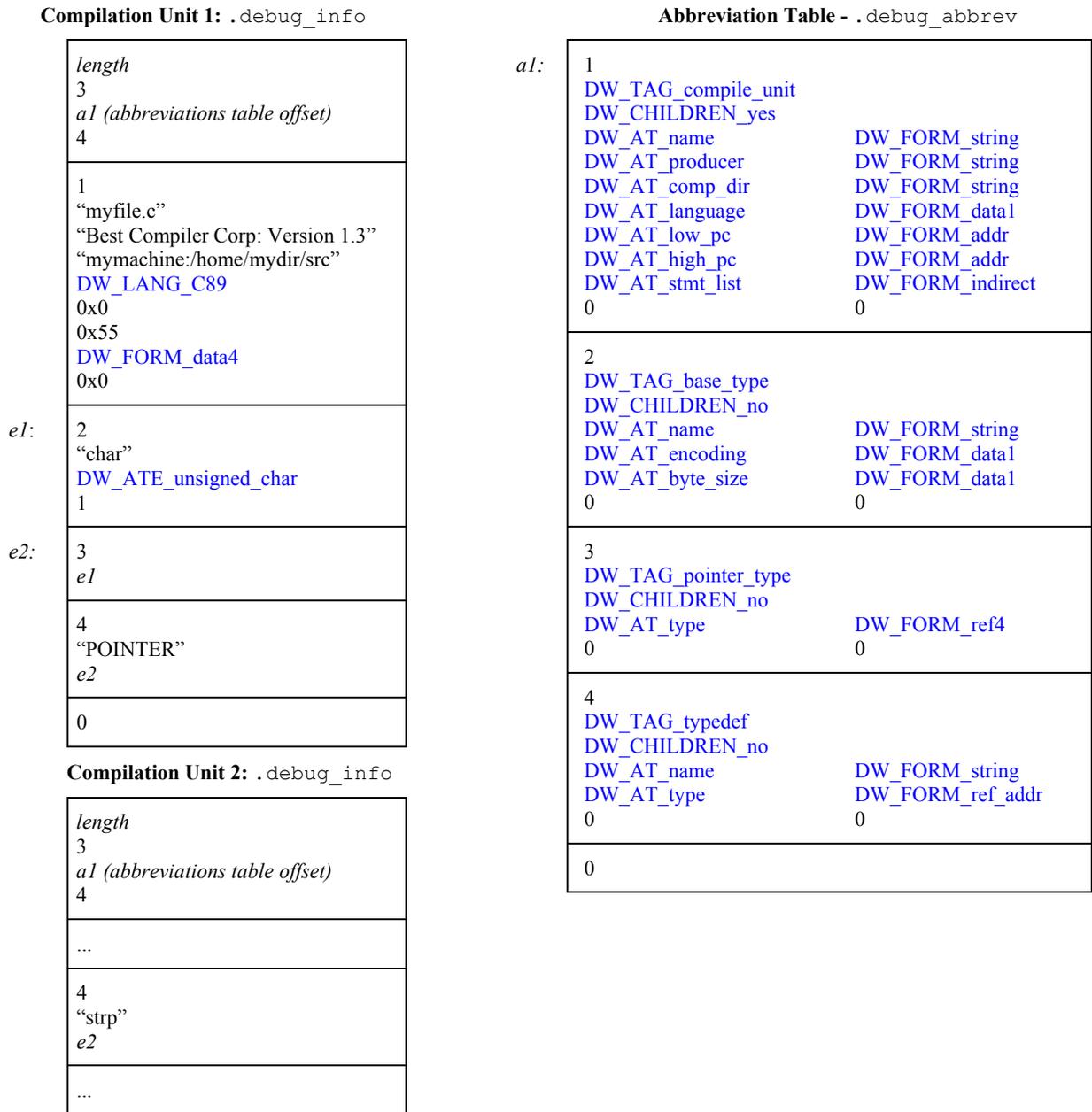


Figure 48. Compilation units and abbreviations table

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.

DWARF Debugging Information Format, Version 3

```
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 an `DW_AT_data_location` attribute. In that case the object doubles as its own descriptor.)

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

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

Appendix -- Examples (informative)

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](#).

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

DWARF Debugging Information Format, Version 3

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

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

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

Appendix -- Examples (informative)

```
8$:      DW_TAG_variable
         DW_AT_name("arrays")
         DW_AT_type(reference to 6$)
         DW_AT_location(machine=
           ...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 stack machine, whose result is the address needed. The object address used in this case is the object we are working on, namely the variable named `arrays`, whose address was found in step 1. (Had there been no `DW_AT_data_location` attribute, the desired address would be the same as the address from step 1.)

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

DWARF Debugging Information Format, Version 3

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.

Appendix -- Examples (informative)

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](#).

DWARF Debugging Information Format, Version 3

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 base type INTEGER)

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

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

      . . .

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

Appendix -- Examples (informative)

```
. . .

26$: DW_TAG_structure_type
    DW_AT_name("REC2")
27$: DW_TAG_member
    DW_AT_name("N")
    DW_AT_type(reference to subtype TEENY at 21$)
    DW_AT_data_member_location(constant 0)
28$: DW_TAG_array_type
    ! No name, default (Ada) order, default stride
    ! Default data location
    DW_AT_TYPE(reference to base type INTEGER)
29$: DW_TAG_subrange_type
    DW_AT_type(reference to subrange TEENY at 21$)
    DW_AT_lower_bound(constant 1)
    DW_AT_upper_bound(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$: DW_TAG_variable
    DW_AT_name("OBJ2B")
    DW_AT_type(reference to type REC2 at 26$)
    DW_AT_location(...as appropriate...)
```

Figure 53. Ada example: DWARF description

DWARF Debugging Information Format, Version 3

D.3 Namespace Examples

The C++ example in [Figure 54](#) 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 54. Namespace example: source fragment

Appendix -- Examples (informative)

The DWARF representation in [Figure 55](#) is appropriate.

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

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

DWARF Debugging Information Format, Version 3

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

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

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

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

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

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

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

Figure 55. Namespace example: DWARF description

D.4 Member Function Example

Consider the member function example fragment in [Figure 56](#).

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

Figure 56. Member function example: source fragment

The DWARF description in [Figure 57](#) 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$)
        ...
```

DWARF Debugging Information Format, Version 3

```
7$: DW_TAG_subprogram
    DW_AT_name("func1")
    DW_AT_type(reference to 1$)
    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_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$: DW_TAG_subprogram
    DW_AT_name("func3")
    DW_AT_type(reference to 1$)
    ...
    ! No 'this' formal parameter => func3 is static
13$: DW_TAG_formal_parameter
    DW_AT_name(x3)
    DW_AT_type(reference to 2$)
    ...
```

Figure 57. Member function example: DWARF description

Appendix -- Examples (informative)

D.5 Line Number Program Example

Consider the simple source file and the resulting machine code for the Intel 8086 processor in [Figure 58](#).

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

Suppose the line number program header specifies the following:

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

DWARF Debugging Information Format, Version 3

Figure 59 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
<code>DW_LNS_advance_pc</code>	LEB128 (0x239)	0x2, 0xb9, 0x04
<code>SPECIAL(2, 0)</code>		0xb
<code>SPECIAL(2, 3)</code>		0x38
<code>SPECIAL(1, 8)</code>		0x82
<code>SPECIAL(1, 7)</code>		0x73
<code>DW_LNS_advance_pc</code>	LEB128 (2)	0x2, 0x2
<code>DW_LNE_end_sequence</code>		0x0, 0x1, 0x1

Figure 59. Line number program example: one encoding

Figure 60 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
<code>DW_LNS_fixed_advance_pc</code>	0x239	0x9, 0x39, 0x2
<code>SPECIAL(2, 0)</code>		0xb
<code>DW_LNS_fixed_advance_pc</code>	0x3	0x9, 0x3, 0x0
<code>SPECIAL(2, 0)</code>		0xb
<code>DW_LNS_fixed_advance_pc</code>	0x8	0x9, 0x8, 0x0
<code>SPECIAL(1, 0)</code>		0xa
<code>DW_LNS_fixed_advance_pc</code>	0x7	0x9, 0x7, 0x0
<code>SPECIAL(1, 0)</code>		0xa
<code>DW_LNS_fixed_advance_pc</code>	0x2	0x9, 0x2, 0x0
<code>DW_LNE_end_sequence</code>		0x0, 0x1, 0x1

Figure 60. 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.

DWARF Debugging Information Format, Version 3

```

;; 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 61. Call frame information example: machine code fragments

The abstract table (see Section 6.4.1) for the foo subroutine is shown in [Figure 62](#). Corresponding fragments from the `.debug_frame` section are shown in [Figure 63](#).

<u>Location</u>	<u>CFA</u>	<u>R0</u>	<u>R1</u>	<u>R2</u>	<u>R3</u>	<u>R4</u>	<u>R5</u>	<u>R6</u>	<u>R7</u>	<u>R8</u>
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 62. Call frame information example: conceptual matrix

Appendix -- Examples (informative)

The following notes apply to [Figure 62](#):

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

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

Figure 63. Call frame information example: common information entry encoding

DWARF Debugging Information Format, Version 3

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> (2 nd 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> (2 nd parameter)
fde+31	DW_CFA_advance_loc(11)	44/<caf>
fde+32	DW_CFA_restore(4)	
fde+33	DW_CFA_advance_loc(1)	
fde+34	DW_CFA_restore(6)	
fde+35	DW_CFA_def_cfa_register(7)	
fde+37	DW_CFA_advance_loc(1)	
fde+38	DW_CFA_restore(8)	
fde+39	DW_CFA_advance_loc(1)	
fde+40	DW_CFA_def_cfa_offset(0)	
fde+42	DW_CFA_nop	padding
fde+43	DW_CFA_nop	padding
fde+44		

Figure 64. Call frame information example: frame description entry encoding

D.7 Inlining Examples

The pseudo-source in [Figure 65](#) 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 65. 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.)

DWARF Debugging Information Format, Version 3

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 66](#).

Notice in [Figure 66](#) 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 65](#) might be described as shown in [Figure 67](#).

Appendix -- Examples (informative)

```
! Abstract instance for OUTER
!
OUTER.AI.1.1.1:
  DW_TAG_subprogram
    DW_AT_name("OUTER")
    DW_AT_inline(DW_INL_declared_inlined)
    ! No low/high PCs
OUTER.AI.1.1.2:
  DW_TAG_formal_parameter
    DW_AT_name("OUTER_FORMAL")
    DW_AT_type(reference to integer)
    ! No location
OUTER.AI.1.1.3:
  DW_TAG_variable
    DW_AT_name("OUTER_LOCAL")
    DW_AT_type(reference to integer)
    ! No location
!
! Abstract instance for INNER
!
INNER.AI.1.1.1:
  DW_TAG_subprogram
    DW_AT_name("INNER")
    DW_AT_inline(DW_INL_declared_inlined)
    ! No low/high PCs
INNER.AI.1.1.2:
  DW_TAG_formal_parameter
    DW_AT_name("INNER_FORMAL")
    DW_AT_type(reference to integer)
    ! No location
INNER.AI.1.1.3:
  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 66. Inlining example #1: abstract instance

DWARF Debugging Information Format, Version 3

```
! 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 67. Inlining example #1: concrete instance

Appendix -- Examples (informative)

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 inline instance of OUTER has its own normal instance of INNER. The abstract instance for OUTER, which includes INNER, is shown in [Figure 68](#).

Note that the debugging information in this Figure differs from that in [Figure 66](#) in that INNER lacks a [DW_AT_inline](#) attribute and therefore is not a distinct abstract instance. INNER is merely an out-of-line routine that is part of OUTER's abstract instance. This is reflected in the Figure by the fact that the labels for INNER use the substring OUTER instead of INNER.

A resulting concrete inline instance of OUTER is shown in [Figure 69](#).

Notice in [Figure 69](#) 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.

DWARF Debugging Information Format, Version 3

```
! 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 68. Inlining example #2: abstract instance

Appendix -- Examples (informative)

```
! 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.AI.2.6)  
    DW_AT_location(...)  
...  
0  
...  
0
```

Figure 69. Inlining example #2: concrete instance

DWARF Debugging Information Format, Version 3

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 inline instances of OUTER. The abstract instance for OUTER is shown in [Figure 70](#).

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

A concrete inline instance of OUTER is illustrated in [Figure 71](#).

Notice in [Figure 71](#) that there is no DWARF representation for INNER at all; the representation of INNER does not vary across instances of OUTER and the abstract instance of OUTER includes the complete description of INNER, so that the description of INNER may be (and for reasons of space efficiency, should be) omitted from each concrete instance of OUTER.

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

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

Appendix -- Examples (informative)

```
! 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 70. Inlining example #3: abstract instance

DWARF Debugging Information Format, Version 3

```
! 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 71. Inlining example #3: concrete instance

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 Overview

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

The following uses some traditional section naming here but aside from the DWARF sections, the names are just 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
```

DWARF Debugging Information Format, Version 3

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.
3. 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.

Appendix -- DWARF Compression and Duplicate Elimination (informative)

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

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

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

E.2 Naming and Usage Considerations

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

Implementations should clearly document their naming conventions.

In the following, it will be helpful to refer to the examples in [Figure 72](#) through [Figure 79](#) of [Section E.3](#).

DWARF Debugging Information Format, Version 3

E.2.1 Section Group Names

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

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

will suffice, where

- `<producer-prefix>` is some string specific to the producer, which has a language-designation embedded in the name when appropriate. (Alternatively, the language name could be embedded in the `<gid-number>`).
- `<file-designator>` names the file, such as `wa.h` in the example.
- `<gid-number>` is a string generated to identify the specific `wa.h` header file in such a way that
 - a 'matching' output from another compile generates the same `<gid-number>`, and
 - a non-matching output (say because of `#defines`) generates a different `<gid-number>`.

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

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

E.2.2 Debugging Information Entry Names

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

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

such as

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

Appendix -- DWARF Compression and Duplicate Elimination (informative)

where

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

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

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

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

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

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

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

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

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

E.2.3 Use of `DW_TAG_compile_unit` versus `DW_TAG_partial_unit`

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

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

DWARF Debugging Information Format, Version 3

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

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

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

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

E.2.4 Use of `DW_TAG_imported_unit`

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

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

E.2.5 Use of `DW_FORM_ref_addr`

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

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

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

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

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

E.3 Examples

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

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

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

E.3.1 C++ Example

The C++ source in [Figure 72](#) 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 72. Duplicate elimination example #1: C++ source

DWARF Debugging Information Format, Version 3

Figure 73 shows the section group corresponding to the included file `wa.h`.

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

== section .debug_info

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

Figure 73. Duplicate elimination example #1: DWARF section group

Appendix -- DWARF Compression and Duplicate Elimination (informative)

Figure 74 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 74. 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.

DWARF duplicate elimination for C++ requires only DWARF debugging information entries and attributes that are defined in DWARF Version 2, together with linker support for section groups.

DWARF Debugging Information Format, Version 3

E.3.2 Fortran Example

For a Fortran example, consider [Figure 75](#).

```
---- 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 75. Duplicate elimination example #2: Fortran source

[Figure 76](#) shows the section group corresponding to the included file `CommonStuff.fh`.

Appendix -- DWARF Compression and Duplicate Elimination (informative)

```
==== 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 76. Duplicate elimination example #2: DWARF section group

DWARF Debugging Information Format, Version 3

Figure 77 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 77. Duplicate elimination example #2: primary unit

A companion main program is shown in Figure 78.

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

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

Figure 78. Duplicate elimination example #2: companion source

Appendix -- DWARF Compression and Duplicate Elimination (informative)

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 79](#).

```
== 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 79. Duplicate elimination example #2: companion DWARF

This example uses [DW_TAG_partial_unit](#) for the section group because the included declarations are not independently visible as global entities.

E.3.3 C Example

The C++ example in [Section E.3.1](#) might appear to be equally valid as a C example. However, it is prudent to include a [DW_TAG_imported_unit](#) in the primary unit (see [Figure 74](#)) 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.

DWARF Debugging Information Format, Version 3

E.4 Summary of Compression Techniques

E.4.1 #include compression

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

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

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

E.4.2 Eliminating function duplication

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

For each such code group (function template in this example) the compiler assigns a name for the group which will match all other instantiations of this function but match nothing else. The 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.4.3 Single-function-per-DWARF-compilation-unit

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

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

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

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

Appendix -- DWARF Compression and Duplicate Elimination (informative)

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

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

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

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

DWARF Debugging Information Format, Version 3

DWARF Debugging Information Format, Version 3

Appendix F – 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 80](#) 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` section and so an incompatible change in those sections' format would be represented by a change in the `.debug_info` section version number.

Section Name	Section version number in DWARF Version 2 (July 1993)	Section version number in DWARF Version 3 (this document)
<code>.debug_abbrev</code>	-	-
<code>.debug_aranges</code>	2	2
<code>.debug_frame</code>	1	3
<code>.debug_info</code>	2	3
<code>.debug_line</code>	2	3
<code>.debug_loc</code>	-	-
<code>.debug_macinfo</code>	-	-
<code>.debug_pubnames</code>	2	2
<code>.debug_pubtypes</code>	x	2
<code>.debug_ranges</code>	x	-
<code>.debug_str</code>	-	-

Figure 80. 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.

For `.debug_frame`, section version 2 is unused.

Higher numbers are reserved for future use.

DWARF Debugging Information Format, Version 3

DWARF Debugging Information Format, Version 3

GNU Free Documentation License

Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc.

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

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-

DWARF Debugging Information Format, Version 3

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

GNU Free Documentation License

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.

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.

DWARF Debugging Information Format, Version 3

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

GNU Free Documentation License

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.

DWARF Debugging Information Format, Version 3

- 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.

GNU Free Documentation License

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.

DWARF Debugging Information Format, Version 3

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 for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

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

GNU Free Documentation License

choose any version ever published (not as a draft) by the Free Software Foundation.

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.

DWARF Debugging Information Format, Version 3

DWARF Debugging Information Format, Version 3

Index

... parameters.....	<i>See</i> unspecified parameters entry
.data	219
.debug_abbrev.....	121, 123, 124, 160, 183, 219, 235
example	187
.debug_aranges	91, 120, 121, 154, 160, 183, 219, 235
.debug_frame	109, 111, 120, 121, 160, 235
example	206
.debug_info.5, 89, 90, 91, 92, 120, 121, 122, 123, 124, 125, 128, 154, 160, 183, 219, 221, 223, 224, 225, 227, 229, 230, 231, 232, 235	
example	187
.debug_line	39, 92, 120, 121, 127, 160, 183, 219, 235
.debug_loc.....	26, 127, 160, 183, 235
.debug_macinfo	39, 104, 106, 128, 160, 183, 235
.debug_pubnames	90, 120, 121, 122, 153, 160, 183, 235
.debug_pubtypes	90, 120, 121, 122, 153, 160, 183, 235
.debug_ranges	33, 128, 160, 183, 235
.debug_str	122, 129, 160, 183, 235
.text.....	219, 227, 230
32-bit DWARF format.....	120
64-bit DWARF format.....	121
abbreviations table.....	123, 124
dynamic forms in.....	125
example	187
abstract instance.....	233
example	210, 213, 216
nested.....	55
abstract instance entry.....	51
abstract instance root	51
abstract instance tree.....	51, 52
abstract origin attribute	53, 54
accelerated access	89
by address.....	91
by name	90
access declaration entry	74
accessibility attribute	28, 73, 74, 77
encoding	148
activation, call frame.....	107
Ada	7, 28, 35, 38, 66, 67, 68, 87, 190, 195, 196, 197
address	
dereference operator	17
implicit push for member pointer	85
implicit push of base.....	18, 25
size of an.....	<i>See</i> size of an address
address class	12, 126
address class attribute.....	30, 48, 67
encoding.....	151
address range	
in location list.....	27
in range list.....	34
address register	
in call frame information.....	109
in line number machine.....	93
address size	<i>See</i> size of an address
address space	
flat30	
multiple	17
segmented	29, 91, 123, 155
address, uplevel.....	<i>See</i> static link attribute
alias declaration.....	<i>See</i> imported declaration entry
allocated attribute	86
anonymous object containing bitfield.....	75
anonymous union	59, 74
ARM instruction set architecture.....	92
array	
declaration of type.....	69
descriptor for.....	189
element ordering	69
element type	70
array type entry	69
examples	189
artificial attribute	29
associated attribute	86
attribute duplication.....	5
attribute encodings	125
attribute ordering.....	5
attribute value classes.....	<i>See</i> static link attribute
attributes.....	5
list of.....	6
base address selection entry	
in location list.....	26, 27, 146
in range list.....	33, 34, 159
base type entry.....	63
base types attribute.....	41
big endian byte order	
effect of.....	76
big-endian encoding	<i>See</i> endianness attribute
binary scale attribute	66
bit offset attribute	63, 75
bit size attribute	63, 75

DWARF Debugging Information Format, Version 3

bit stride attribute.....	70, 81, 84	containing type attribute.....	85
block class.....	12, 126	contiguous address range.....	33
block entry.....	<i>See</i> try block entry, <i>See</i> lexical block entry	count attribute.....	67, 84
builtin type.....	<i>See</i> base type entry	default.....	84
byte size attribute.....	63, 70, 71, 75, 81, 82, 83, 86	D 38	
byte stride attribute.....	81	data location attribute.....	86
C 1, 3, 31, 38, 47, 57, 59, 61, 64, 67, 68, 69, 70, 71, 72,		data member.....	<i>See</i> member entry (data)
81, 82, 84, 104, 189, 190, 231, 232		data member location attribute.....	73, 74
C++...1, 3, 28, 29, 32, 38, 42, 43, 44, 45, 49, 50, 54, 56, 57,		debugging information entry.....	5
58, 59, 60, 67, 68, 69, 70, 71, 72, 73, 74, 77, 78, 81, 84,		global name for.....	222
89, 90, 104, 216, 219, 222, 225, 227, 231, 232		ownership relation.....	13
call column attribute.....	52	decimal scale attribute.....	65, 66
call file attribute.....	52	decimal sign attribute.....	65
call frame information		declaration attribute.....	31, 42, 59, 71
encoding.....	157	declaration column attribute.....	32
examples.....	205	declaration coordinates.....	31
in concrete instance.....	53	in concrete instance.....	53
call line attribute.....	52	declaration file attribute.....	31
calling convention attribute.....	46	declaration line attribute.....	31
encoding.....	152	default value attribute.....	60
case sensitivity.....	40	derived type (C++).....	<i>See</i> inheritance entry
catch block entry.....	58	description attribute.....	35
CIE.....	<i>See</i> common information entry	descriptor, array.....	189
class template instantiation (entry).....	78	digit count attribute.....	65,
class type entry.....	71	discontiguous address ranges ...	<i>See</i> non-contiguous address
as class template instantiation.....	78	ranges	
classes of attribute value.....	5, <i>See also</i> attribute encodings	discriminant (entry).....	79
COBOL.....	65	discriminant attribute.....	79
column position of declaration.....	32	discriminant list attribute.....	79
COMDAT.....	<i>See</i> section group	encoding.....	153
common (block) reference attribute.....	48	discriminant value attribute.....	79
common block.....	<i>See</i> Fortran, common block	duplicate elimination....	<i>See</i> DWARF duplicate elimination
common block entry.....	62	DW_ACCESS_private.....	28, 148
common information entry.....	110	DW_ACCESS_protected.....	28, 148
compilation directory attribute.....	40	DW_ACCESS_public.....	28, 148
compilation unit.....	37	DW_ADDR_far16.....	30
for template instantiation.....	78	DW_ADDR_far32.....	30
header.....	123	DW_ADDR_huge16.....	30
normal.....	37	DW_ADDR_near16.....	30
partial.....	37	DW_ADDR_near32.....	30
compression.....	<i>See</i> DWARF compression	DW_ADDR_none.....	30, 151
concrete inline instance		DW_AT_abstract_origin 7, 53, 54, 135, 163, 164, 167, 168,	
example.....	210, 213, 216	169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179,	
nested.....	55	212, 215, 218	
concrete inlined instance root.....	52	DW_AT_accessibility 7, 28, 73, 74, 77, 135, 148, 163, 164,	
concrete inlined instance tree.....	52	166, 167, 168, 169, 170, 171, 173, 174, 175, 176, 177,	
concrete out-of-line instance.....	54, 233	178, 179	
example.....	213	DW_AT_address_class ... 7, 30, 48, 67, 135, 166, 172, 173,	
of inlined subprogram.....	54	175, 176, 179	
condition entry.....	80	DW_AT_allocated 7, 35, 70, 86, 87, 136, 163, 164, 165,	
condition, COBOL level-88.....	80	167, 171, 172, 173, 174, 176, 177, 178, 179, 192	
constant class.....	12, 126	DW_AT_artificial 7, 29, 56, 77, 135, 168, 175, 178, 202	
constant entry.....	59	DW_AT_associated ... 7, 35, 70, 86, 87, 136, 163, 164, 165,	
constant type entry.....	68	167, 171, 172, 173, 174, 176, 177, 178, 179, 191	
constant value attribute.....	61, 78, 81		

Index

- DW_AT_base_types..... 7, 41, 135, 165, 172
- DW_AT_binary_scale..... 7, 66, 137, 164
- DW_AT_bit_offset..... 7, 35, 63, 75, 133, 164, 170
- DW_AT_bit_size..... 7, 35, 63, 75, 133, 164, 170
- DW_AT_bit_stride..... 7, 35, 70, 81, 84, 135, 163, 167, 176
- DW_AT_byte_size..... 7, 35, 63, 70, 71, 75, 81, 82, 83, 86, 133, 163, 164, 167, 170, 173, 174, 176, 188, 192, 231
- DW_AT_byte_stride..... 7, 35, 81, 84, 136, 167, 176, 194
- DW_AT_call_column..... 7, 52, 137, 169
- DW_AT_call_file..... 7, 52, 137, 169
- DW_AT_call_line..... 7, 52, 137, 169
- DW_AT_calling_convention..... 7, 46, 135, 152, 175
- DW_AT_common_reference..... 8, 48, 134, 165, 230, 231
- DW_AT_comp_dir..... 8, 40, 134, 165, 172, 188
- DW_AT_const_value..... 8, 61, 78, 81, 134, 166, 167, 177, 178, 212, 229
- DW_AT_containing_type..... 8, 85, 134, 172
- DW_AT_count..... 8, 35, 67, 84, 135, 173, 176
- DW_AT_data_location.. 8, 70, 86, 136, 163, 164, 165, 167, 171, 172, 173, 174, 176, 177, 178, 179, 190, 191, 192, 193, 194, 196
- DW_AT_data_member_location..... 8, 18, 73, 74, 135, 169, 170, 192, 197
- DW_AT_decimal_scale..... 8, 65, 66, 164
- DW_AT_decimal_sign..... 8, 65, 137, 164
- DW_AT_decl_column..... 8, 31, 32, 135, 163
- DW_AT_decl_file..... 8, 31, 135, 163
- DW_AT_decl_line..... 8, 31, 135, 163
- DW_AT_declaration.... 8, 31, 42, 59, 71, 72, 135, 163, 164, 165, 166, 167, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179
- DW_AT_default_value..... 8, 60, 134, 168
- DW_AT_description.... 8, 35, 137, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178
- DW_AT_digit_count..... 8, 65, 137, 164
- DW_AT_discr..... 8, 79, 80, 134, 135, 153, 179
- DW_AT_discr_list..... 8, 79, 80, 135, 153, 179
- DW_AT_discr_value..... 8, 79, 80, 134, 179
- DW_AT_elemental..... 8, 46, 137, 175
- DW_AT_encoding..... 9, 63, 135, 146, 164, 188, 231
- DW_AT_endianity..... 9, 61, 63, 137, 164, 166, 168, 178
- DW_AT_entry_pc ... 9, 29, 33, 34, 42, 47, 51, 52, 136, 169, 171, 175
- DW_AT_explicit..... 9, 77, 137, 175
- DW_AT_extension..... 9, 42, 136, 171, 200
- DW_AT_external..... 9, 46, 59, 135, 166, 175, 178
- DW_AT_frame_base. 9, 16, 25, 48, 49, 135, 166, 175, 215, 217, 218
- DW_AT_friend..... 9, 74, 135, 168, 178
- DW_AT_hi_user..... 138
- DW_AT_high_pc .. 9, 29, 32, 33, 38, 42, 47, 51, 52, 57, 58, 134, 164, 165, 169, 170, 171, 172, 175, 177, 179, 188, 199, 200, 212, 215, 217, 218
- DW_AT_identifier_case..... 9, 40, 136, 151, 165, 172, 229
- DW_AT_import..... 9, 43, 44, 134, 168, 200, 224, 230, 231
- DW_AT_inline 9, 50, 51, 134, 152, 175, 211, 213, 214, 217
- DW_AT_is_optional..... 9, 60, 134, 168
- DW_AT_language .. 9, 38, 69, 134, 165, 172, 188, 226, 229
- DW_AT_lo_user..... 138
- DW_AT_location..... 9, 21, 32, 51, 58, 59, 62, 87, 133, 165, 166, 168, 178, 179, 193, 197, 199, 200, 202, 212, 215, 217, 218, 229
- DW_AT_low_pc.... 9, 29, 32, 33, 34, 38, 42, 47, 51, 52, 57, 58, 134, 164, 165, 166, 169, 170, 171, 172, 175, 177, 179, 188, 199, 200, 212, 215, 217, 218
- DW_AT_lower_bound... 9, 35, 84, 134, 176, 191, 192, 196, 197, 229
- DW_AT_macro_info..... 10, 39, 136, 165, 172
- DW_AT_mutable..... 10, 74, 137, 170
- DW_AT_name10, 32, 35, 38, 40, 42, 43, 45, 50, 54, 57, 59, 62, 63, 67, 69, 71, 72, 74, 78, 80, 81, 82, 83, 85, 86, 90, 133, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 188, 192, 193, 196, 197, 199, 200, 201, 202, 211, 214, 217, 226, 227, 229, 230, 231
- DW_AT_namelist_item..... 10, 62, 136, 171
- DW_AT_object_pointer..... 10, 77, 137, 175, 202
- DW_AT_ordering..... 10, 69, 133, 153, 163
- DW_AT_picture_string..... 10, 65, 137, 164
- DW_AT_priority..... 10, 42, 136, 171
- DW_AT_producer..... 10, 40, 134, 165, 172, 188
- DW_AT_prototyped..... 10, 47, 82, 134, 175, 176
- DW_AT_pure..... 10, 47, 137, 175
- DW_AT_ranges ... 10, 29, 32, 33, 38, 42, 47, 51, 52, 57, 58, 136, 164, 165, 169, 170, 171, 172, 175, 177, 179
- DW_AT_recursive..... 10, 47, 138, 175
- DW_AT_return_addr..... 10, 48, 51, 134, 166, 169, 175
- DW_AT_segment 10, 29, 48, 51, 60, 136, 164, 165, 166, 168, 169, 170, 171, 172, 175, 177, 178, 179
- DW_AT_sibling. 10, 13, 133, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179
- DW_AT_small..... 66, 137, 164
- DW_AT_specification 10, 43, 60, 72, 77, 78, 136, 164, 174, 175, 178, 200
- DW_AT_start_scope..... 10, 51, 61, 63, 135, 163, 164, 166, 167, 168, 169, 171, 173, 174, 175, 176, 177, 178
- DW_AT_static_link 10, 49, 136, 166, 175, 212, 215, 217
- DW_AT_stmt_list..... 11, 39, 134, 165, 172, 188
- DW_AT_stride..... *See DW_AT_byte_stride, See DW_AT_byte_stride*
- DW_AT_stride_size..... *See DW_AT_bit_stride*
- DW_AT_string_length..... 11, 82, 134, 174
- DW_AT_threads_scaled..... 11, 83, 137, 176
- DW_AT_trampoline..... 11, 56, 137, 169, 175
- DW_AT_type. 11, 27, 47, 50, 58, 60, 67, 69, 70, 73, 74, 78, 79, 80, 81, 82, 83, 85, 86, 136, 163, 165, 166, 167, 168, 169, 170, 171, 172, 173, 175, 176, 177, 178, 179, 188,

DWARF Debugging Information Format, Version 3

191, 192, 193, 196, 197, 199, 200, 201, 202, 211, 214, 217, 226, 227, 229, 230	DW_CFA_restore_state	117, 158
DW_AT_upper_bound 11, 35, 84, 135, 176, 191, 192, 196, 197, 229	DW_CFA_same_value.....	114, 158, 207
DW_AT_use_location	DW_CFA_set_loc	112, 117, 158
DW_AT_use_UTF8	DW_CFA_undefined	114, 118, 158, 207
DW_AT_variable_parameter.....	DW_CFA_val_expression.....	116, 159
DW_AT_virtuality.....	DW_CFA_val_offset	115, 159
DW_AT_visibility.....	DW_CFA_val_offset_sf.....	115
167, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179	DW_CHILDREN_no	124, 133, 188
DW_AT_vtable_elem_location	DW_CHILDREN_yes.....	124, 133, 188
DW_ATE_address.....	DW_DS_leading_overpunch.....	66, 147
DW_ATE_boolean	DW_DS_leading_separate	66, 147
DW_ATE_complex_float.....	DW_DS_trailing_overpunch.....	66, 147
DW_ATE_decimal_float.....	DW_DS_trailing_separate.....	66, 147
DW_ATE_edited.....	DW_DS_unsigned.....	66, 147
DW_ATE_float.....	DW_DSC_label.....	79, 153
DW_ATE_hi_user.....	DW_DSC_range	79, 153
DW_ATE_imaginary_float.....	DW_END_big	62, 148
DW_ATE_lo_user.....	DW_END_default.....	62, 148
DW_ATE_numeric_string.....	DW_END_hi_user	148
DW_ATE_packed_decimal.....	DW_END_little.....	62, 148
DW_ATE_signed	DW_END_lo_user	148
DW_ATE_signed_char.....	DW_FORM_addr.....	126, 138, 146, 159, 188
DW_ATE_signed_fixed	DW_FORM_block.....	114, 116, 126, 138
DW_ATE_unsigned	DW_FORM_block1	126, 138
DW_ATE_unsigned_char.....	DW_FORM_block2.....	126, 138
DW_ATE_unsigned_fixed	DW_FORM_block4.....	126, 138
DW_CC_hi_user.....	DW_FORM_data1	126, 138, 188
DW_CC_lo_user.....	DW_FORM_data2	126, 138
DW_CC_nocall.....	DW_FORM_data4 ..	122, 125, 126, 127, 128, 138, 183, 188
DW_CC_normal.....	DW_FORM_data8	122, 125, 126, 127, 128, 138
DW_CC_program.....	DW_FORM_flag.....	127, 138
DW_CFA_advance_loc.....	DW_FORM_indirect.....	125, 139, 188
DW_CFA_advance_loc1	DW_FORM_ref_addr21, 122, 128, 139, 221, 223, 224, 233	
DW_CFA_advance_loc2	DW_FORM_ref_udata.....	128, 139, 223
DW_CFA_advance_loc4	DW_FORM_ref1	128, 139, 223
DW_CFA_def_cfa.....	DW_FORM_ref2	21, 128, 139, 223
DW_CFA_def_cfa_expression.....	DW_FORM_ref4	21, 128, 139, 188, 223
DW_CFA_def_cfa_offset.....	DW_FORM_ref8	128, 139, 223
DW_CFA_def_cfa_offset_sf.....	DW_FORM_sdata.....	126, 138, 139, 140, 141
DW_CFA_def_cfa_register.....	DW_FORM_string.....	129, 138, 188
DW_CFA_def_cfa_sf.....	DW_FORM_strp.....	122, 125, 129, 139, 183
DW_CFA_expression.....	DW_FORM_udata	126, 139, 140
DW_CFA_hi_user	DW_ID_case_insensitive.....	40, 41, 151, 229
DW_CFA_lo_user.....	DW_ID_case_sensitive	40, 151
DW_CFA_nop.....	DW_ID_down_case	40, 151
DW_CFA_offset.....	DW_ID_up_case	40, 151
DW_CFA_offset_extended.....	DW_INL_declared_inlined.....	51, 152, 211, 214, 217
DW_CFA_offset_extended_sf.....	DW_INL_declared_not_inlined.....	51, 152
DW_CFA_register.....	DW_INL_inlined	51, 152
DW_CFA_remember_state.....	DW_INL_not_inlined	51, 152
DW_CFA_restore	DW_LANG_Ada83	38, 150
DW_CFA_restore_extended.....	DW_LANG_Ada95	38, 150
	DW_LANG_C	38, 150, 188
	DW_LANG_C_plus_plus	38, 150, 226

Index

DW_LANG_C89.....	38, 150, 188	DW_OP_call4.....	21, 87, 112, 114, 116, 145
DW_LANG_C99.....	38, 150	DW_OP_const1s.....	15, 142
DW_LANG_Cobol74.....	38, 150	DW_OP_const1u.....	15, 142
DW_LANG_Cobol85.....	38, 150	DW_OP_const2s.....	15, 142
DW_LANG_D.....	38, 151	DW_OP_const2u.....	15, 142
DW_LANG_Fortran77.....	38, 150	DW_OP_const4s.....	15, 142
DW_LANG_Fortran90.....	38, 150, 229	DW_OP_const4u.....	15, 142
DW_LANG_Fortran95.....	39, 150	DW_OP_const8s.....	15, 142
DW_LANG_hi_user.....	151	DW_OP_const8u.....	15, 142
DW_LANG_Java.....	39, 150	DW_OP_consts.....	15, 142
DW_LANG_lo_user.....	151	DW_OP_constu.....	15, 142
DW_LANG_Modula2.....	39, 150	DW_OP_deref.....	17, 26, 142, 191, 192
DW_LANG_ObjC.....	39, 151	DW_OP_deref_size.....	17, 145
DW_LANG_ObjC_plus_plus.....	39, 151	DW_OP_div.....	19, 143
DW_LANG_Pascal83.....	39, 150	DW_OP_drop.....	16, 22, 142
DW_LANG_PLI.....	39, 150	DW_OP_dup.....	16, 22, 142
DW_LANG_UPC.....	39, 151	DW_OP_eq.....	20, 143
DW_LNE_define_file.....	98, 103, 104, 156	DW_OP_fbreg.....	16, 25, 26, 144
DW_LNE_end_sequence.....	103, 156, 204	DW_OP_form_tls_address.....	18, 145
DW_LNE_hi_user.....	156	DW_OP_ge.....	20, 143
DW_LNE_lo_user.....	156	DW_OP_gt.....	20, 144
DW_LNE_set_address.....	103, 156	DW_OP_hi_user.....	145
DW_LNS_advance_line.....	101, 155	DW_OP_le.....	20, 144
DW_LNS_advance_pc.....	100, 101, 102, 155, 204	DW_OP_lit0.....	14, 144
DW_LNS_const_add_pc.....	101, 155	DW_OP_lit1.....	14, 144, 191
DW_LNS_copy.....	100, 155	DW_OP_lit2.....	192
DW_LNS_fixed_advance_pc.....	95, 101, 156, 204	DW_OP_lit31.....	14, 144
DW_LNS_hi_user omission.....	119	DW_OP_lo_user.....	145
DW_LNS_lo_user omission.....	119	DW_OP_lt.....	20, 144
DW_LNS_negate_stmt.....	96, 101, 155	DW_OP_minus.....	19, 143
DW_LNS_set_basic_block.....	101, 155	DW_OP_mod.....	19, 143
DW_LNS_set_column.....	101, 155	DW_OP_mul.....	19, 143
DW_LNS_set_epilogue_begin.....	102, 156	DW_OP_ne.....	20, 143, 144
DW_LNS_set_file.....	101, 155	DW_OP_neg.....	19, 143
DW_LNS_set_isa.....	103, 156	DW_OP_nop.....	21, 145
DW_LNS_set_prologue_end.....	102, 156	DW_OP_not.....	19, 143
DW_MACINFO_define.....	104, 105, 106, 157	DW_OP_or.....	19, 143
DW_MACINFO_end_file.....	104, 105, 106, 157	DW_OP_over.....	16, 22, 142
DW_MACINFO_start_file.....	104, 105, 106, 157	DW_OP_pick.....	16, 22, 142
DW_MACINFO_undef.....	104, 105, 106, 157	DW_OP_piece.....	24, 26, 145
DW_MACINFO_vendor_ext.....	104, 106, 157	DW_OP_plus.....	19, 143, 191, 192, 197
DW_OP_abs.....	19, 143	DW_OP_plus_uconst.....	20, 26, 143
DW_OP_addr.....	14, 25, 142	DW_OP_push_object_address.....	18, 75, 86, 112, 114, 116, 145, 191, 192, 193, 194
DW_OP_and.....	19, 143, 191, 192	DW_OP_reg0.....	23, 26, 144
DW_OP_bit_piece.....	24, 145	DW_OP_reg1.....	23, 144
DW_OP_bra.....	21, 143	DW_OP_reg10.....	26
DW_OP_breg0.....	16, 144	DW_OP_reg3.....	26
DW_OP_breg1.....	16, 144	DW_OP_reg31.....	23, 144
DW_OP_breg11.....	25	DW_OP_regx.....	23, 25, 144
DW_OP_breg31.....	16, 144	DW_OP_rot.....	17, 22, 142
DW_OP_bregx.....	16, 26, 144	DW_OP_shl.....	20, 143
DW_OP_call_frame_cfa.....	18, 112, 116, 145	DW_OP_shr.....	20, 143
DW_OP_call_ref.....	21, 87, 112, 114, 116, 145, 183	DW_OP_shra.....	20, 143
DW_OP_call2.....	21, 87, 112, 114, 116, 145		

DWARF Debugging Information Format, Version 3

DW_OP_skip.....	21, 143	DW_TAG_set_type.....	6, 83, 131, 173
DW_OP_swap.....	16, 22, 142	DW_TAG_shared_type.....	6, 68, 133, 173
DW_OP_xderef.....	17, 143	DW_TAG_string_type.....	6, 82, 130, 174
DW_OP_xderef_size.....	17, 145	DW_TAG_structure_type6, 71, 78, 130, 174, 192, 197, 226	
DW_OP_xor.....	20, 143	DW_TAG_subprogram..6, 45, 46, 50, 53, 56, 77, 131, 175, 199, 200, 202, 211, 214, 215, 217, 218, 227, 230, 231	
DW_ORD_col_major.....	69, 153	DW_TAG_subrange_type6, 70, 80, 83, 131, 176, 191, 192, 196, 197, 229	
DW_ORD_row_major.....	69, 153	DW_TAG_subroutine_type.....	6, 81, 130, 176
DW_TAG_access_declaration.....	6, 74, 131, 163	DW_TAG_template_type_parameter.....	6, 50, 78, 132, 177
DW_TAG_array_type..6, 69, 130, 163, 191, 192, 196, 197, 229		DW_TAG_template_value_parameter.....	6, 78, 132, 177
DW_TAG_base_type6, 63, 68, 69, 131, 164, 188, 201, 226, 231		DW_TAG_thrown_type.....	6, 49, 132, 177
DW_TAG_catch_block.....	6, 58, 131, 164	DW_TAG_try_block.....	6, 58, 132, 177
DW_TAG_class_type.....	6, 71, 78, 130, 164, 201	DW_TAG_typedef.....	6, 69, 130, 177, 188
DW_TAG_common_block.....	6, 62, 130, 165, 229	DW_TAG_union_type.....	6, 71, 78, 130, 178
DW_TAG_common_inclusion.....	6, 48, 131, 165, 230, 231	DW_TAG_unspecified_parameters..6, 48, 58, 82, 130, 178	
DW_TAG_compile_unit....6, 37, 123, 130, 165, 188, 223, 224, 226, 227, 230, 231		DW_TAG_unspecified_type.....	6, 67, 132, 178, 201
DW_TAG_condition.....	6, 80, 132, 165	DW_TAG_variable.....	6, 53, 59, 68, 69, 80, 132, 178, 193, 196, 197, 199, 200, 211, 212, 214, 215, 217, 218, 227, 229, 230
DW_TAG_const_type.....	6, 68, 69, 131, 165, 201	DW_TAG_variant.....	6, 79, 130, 132, 179
DW_TAG_constant.....	6, 59, 66, 80, 131, 166, 229	DW_TAG_variant_part.....	6, 79, 132, 179
DW_TAG_dwarf_procedure.....	6, 87, 132, 166	DW_TAG_volatile_type.....	6, 68, 69, 132, 179
DW_TAG_entry_point.....	6, 45, 130, 166	DW_TAG_with_stmt.....	6, 58, 131, 179
DW_TAG_enumeration_type.....	6, 70, 80, 130, 167	DW_VIRTUALITY_none.....	29, 149
DW_TAG_enumerator.....	6, 81, 131, 167	DW_VIRTUALITY_pure_virtual.....	29, 149
DW_TAG_file_type.....	6, 86, 131, 167	DW_VIRTUALITY_virtual.....	29, 149
DW_TAG_formal_parameter..6, 58, 59, 82, 130, 168, 202, 211, 212, 214, 215, 217, 218		DW_VIS_exported.....	28, 149
DW_TAG_friend.....	6, 74, 131, 168	DW_VIS_local.....	28, 149
DW_TAG_hi_user.....	119, 133	DW_VIS_qualified.....	28, 149
DW_TAG_imported_declaration.....	6, 43, 130, 168, 200	DWARF compression.....	219
DW_TAG_imported_module.....	6, 44, 132, 168, 200	DWARF duplicate elimination.....	219
DW_TAG_imported_unit....6, 41, 132, 168, 224, 230, 231, 232		C example.....	231
DW_TAG_inheritance.....	6, 73, 131, 169	C++ example.....	225
DW_TAG_inlined_subroutine.....	6, 45, 52, 53, 55, 56, 131, 169, 211, 212, 215, 218	examples.....	225
DW_TAG_interface_type.....	6, 72, 132, 169	Fortran example.....	228
DW_TAG_label.....	6, 57, 130, 169	DWARF expression.....	14
DW_TAG_lexical_block.....	6, 57, 130, 170	arithmetic operations.....	19
DW_TAG_lo_user.....	119, 133	control flow operations.....	20
DW_TAG_member..6, 60, 74, 80, 130, 170, 192, 196, 197, 226		examples.....	22
DW_TAG_module.....	6, 42, 131, 171	literal encodings.....	14
DW_TAG_namelist.....	6, 62, 131, 171	logical operations.....	19
DW_TAG_namelist_item.....	6, 62, 131, 171	operator encodings.....	142
DW_TAG_namespace.....	6, 42, 132, 171, 199, 200	register name operators.....	23
DW_TAG_packed_type.....	6, 68, 131, 171	special operations.....	21
DW_TAG_partial_unit6, 37, 123, 132, 172, 224, 227, 229, 231		stack operations.....	14
DW_TAG_pointer_type.....	6, 68, 69, 130, 172, 188, 201	DWARF procedure.....	87
DW_TAG_ptr_to_member_type.....	6, 84, 131, 172	DWARF procedure entry.....	87
DW_TAG_reference_type.....	6, 68, 130, 173, 227	DWARF section names, list of.....	160
DW_TAG_restrict_type.....	6, 68, 69, 132, 173	DWARF Version 2 xi, 1, 2, 3, 4, 14, 96, 111, 120, 121, 125, 129, 227, 235	
		elemental attribute.....	46
		encoding attribute.....	63
		encoding.....	146

Index

- end of list entry
 - in location list 27, 146
 - in range list 33, 159
- endianness attribute 61, 63
- entry PC attribute 29
 - and abstract instance 51
 - for inlined subprogram 52
 - for module initialization 42
 - for subroutine 47
- entry point entry 45
- enumeration literal *See* enumerator entry
- enumeration type entry 80
 - as array dimension 70, 81
- enumerator entry 81
- epilogue 94, 98, 100, 102, 107, 108, 117, 156, 206
- error value 120
- exception, thrown *See* thrown type entry
- explicit attribute 77
- extended type (Java) *See* inheritance entry
- extensibility 2
- extension attribute 42
- extension, vendor specific 119
- external attribute 46, 59
- FDE *See* frame description entry
- file containing declaration 31
- file type entry 86
- flag class 12, 127
- formal parameter 48
- formal parameter entry 59, 82
 - in catch block 58
 - with default value 60
- formal type parameter *See* template type parameter entry
- Fortran 1, 3, 38, 45, 47, 62, 82, 84, 86, 87, 189
 - common block 48, 62
 - main program 46
 - module (Fortran 90) 42
 - use statement 44, 45
- frame base attribute 48
- frame description entry 111
- friend attribute 74
- friend entry 74
- function entry *See* subroutine entry
- fundamental type *See* base type entry
- global namespace *See* namespace (C++), global
- hidden indirection *See* data location attribute
- high PC attribute 29, 32, 33, 38, 42, 47, 52, 57, 58
 - and abstract instance 51
- identifier case attribute 40
 - encoding 151
- implementing type (Java) *See* inheritance entry
- import attribute 41, 43, 44
- imported declaration entry 43
- imported module entry 44
- imported unit entry 41
- incomplete class/structure/union 71
- incomplete declaration 31
- inheritance entry 73
- initial length 123
- initial length field 90, 91, 95, 110, 111, 153, 154
 - encoding 120
- inline attribute 50, 51
 - encoding 152
- inlined subprogram call
 - examples 209
- inlined subprogram entry 45, 52
 - in concrete instance 53
- interface type entry 72
- is optional attribute 60
- Java 3, 39, 71, 72, 73
- label entry 57
- language attribute 38, 69
- language name encoding 149
- LEB128
 - examples 140
 - signed, decoding of 186
 - signed, encoding as 139, 185
 - unsigned, decoding of 186
 - unsigned, encoding as 140, 185
- level-88 condition, COBOL 80
- lexical block entry 57
- line number information *See also* statement list attribute
- line number of declaration 31
- line number opcodes
 - extended opcode encoding 156
 - standard opcode encoding 155
- lineptr class 12, 127
- Little Endian Base 128 *See* LEB128
- little endian byte order
 - effect of 76
- little-endian encoding *See* endian attribute
- location attribute 58, 59, 62, 87
 - and abstract instance 51
- location description 23
- location expression 23, 25, *See also* DWARF expression
 - examples 25
 - simple 23
 - use in location list 26
- location list 23, 26, 49, 127, 145, 183
- loclistptr class 12, 127
- lookup
 - by address 91
 - by name 90
- low PC attribute 29, 32, 33, 38, 42, 47, 52, 57, 58
 - and abstract instance 51
- lower bound attribute 84
 - default 84
- macinfo types 104
 - encoding 157

DWARF Debugging Information Format, Version 3

macptr class	12, 128	range list	33, 159, 183
macro formal parameter list	105	rangelistptr class	12, 128
macro information	104	ranges attribute	29, 33, 38, 42, 47, 52, 57, 58
macro information attribute	39	and abstract instance	51
member entry (data)	74	recursive attribute	47
as discriminant	79	reference class	12, 128
member function entry	77	reference type entry	68
MIPS instruction set architecture	92	renamed declaration	<i>See</i> imported declaration entry
Modula2	1, 39, 42, 58	restricted type entry	68
definition module	42	return address attribute	48
module entry	42	and abstract instance	51
mutable attribute	74	return type of subroutine	47
name attribute 32, 35, 38, 40, 42, 43, 45, 50, 54, 57, 59, 62,		sbyte	89, 96, 161
63, 67, 69, 71, 72, 74, 78, 81, 82, 83, 85, 86, 90		section group	220, 223, 225, 226, 228
namelist entry	62	name	222
namelist item attribute	62	section length	
namelist item entry	62	in .debug_aranges header	91
namespace (C++)	42	in .debug_pubnames header	90, 154
alias	44	in .debug_pubtypes header	90, 154
example	198	use in headers	121
global	43	section offset	
std 43		alignment of	160
unnamed	43	in .debug_info header	123
using declaration	44	in .debug_pubnames header	90, 154
using directive	44	in .debug_pubnames offset/name pair	90
namespace declaration entry	42	in .debug_pubtypes header	90
namespace extension entry	42	in .debug_pubtypes name/offset pair	90
nested abstract instance	55	in class lineptr value	127
nested concrete inline instance	55	in class loclistptr value	127
non-contiguous address ranges	33	in class macptr value	128
non-defining declaration	31	in class rangelistptr value	128
normal compilation unit	37	in class reference value	128
object pointer attribute	77	in class string value	129
Objective-C	39, 77	in FDE header	111
Objective-C++	39	in macro information attribute	39
optional parameter	60	in statement list attribute	39
ordering attribute	69	use in headers	121
encoding	153	segment attribute	29, 48
out-of-line instance	<i>See</i> concrete out-of-line instance	and abstract instance	51
packed type entry	68	and data segment	60
parameter	<i>See</i> macro formal parameter list, <i>See</i> this	segmented addressing	<i>See</i> address space
parameter, <i>See</i> variable parameter attribute, <i>See</i> optional		self pointer attribute	<i>See</i> object pointer attribute
parameter attribute, <i>See</i> unspecified parameters entry,		set type entry	83
<i>See</i> template value parameter entry, <i>See</i> template type		shared qualified type	67
parameter entry, <i>See</i> formal parameter entry		shared qualified type entry	68
partial compilation unit	37	sibling attribute	13
Pascal	1, 39, 58, 68, 70, 83, 86	simple location expression	23, 25
pointer to member type entry	84	size of an address 13, 14, 17, 26, 27, 33, 34, 83, 91, 110,	
pointer type entry	68	123, 155	
priority attribute	42	small attribute	66
producer attribute	40	specification attribute	31, 60, 72, 77
prologue	3, 94, 98, 100, 102, 107, 108, 156, 206	start scope attribute	61, 63
prototyped attribute	47, 82	and abstract instance	51
pure attribute	47	statement list attribute	39

Index

- static link attribute49
- stride attribute *See* bit stride attribute or byte stride attribute
- string class 12, 129
- string length attribute82
- string type entry82
- structure type entry71
- subprogram entry45
 - as member function77
 - use for template instantiation.....50
 - use in inlined subprogram.....50
- subrange type entry83
 - as array dimension70
- subroutine type entry81
- tag 5
- tag names *See also* debugging information entry
 - list of.....5
- template instantiation50
 - and special compilation unit78
- template type parameter entry50, 78
- template value parameter entry78
- this parameter29, 56
- this pointer attribute *See* object pointer attribute
- thread-local storage18
- threads scaled attribute83
- thrown exception *See* thrown type entry
- thrown type entry49
- trampoline (subroutine) entry56
- trampoline attribute56
- try block entry58
- type attribute ...27, 47, 50, 58, 60, 67, 69, 70, 73, 74, 78, 79, 82, 83, 85, 86
- 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
- typedef entry69
- ubyte ...89, 91, 95, 96, 97, 98, 100, 110, 111, 112, 113, 123, 155, 161
- uhalf 89, 90, 91, 95, 101, 113, 123, 154, 161
- unallocated variable59
- Unified Parallel C *See* DW_LANG_UPC
- union type entry71
- unnamed namespace *See* namespace (C++), unnamed
- unspecified parameters attribute 48
- unspecified parameters entry 82
 - in catch block 58
- unspecified type entry 67
- UPC68, 83, *See* DW_LANG_UPC
- uplevel address *See* static link attribute
- upper bound attribute 84
 - default 84
- use location attribute 85
- use statement ... *See* Fortran, use statement, *See* Fortran, use statement
- use UTF-8 attribute 41, *See also* UTF-8
- using declaration .. *See* namespace (C++), using declaration
- using directive *See* namespace (C++), using directive
- UTF-83, 11, 41, 110, 129
- uword 89, 113, 161
- variable entry59
 - examples 189
 - in concrete instance 53
- variable length data 139, *See also* LEB128
- variable parameter attribute60
- variant entry79
- variant part entry79
- vendor extensibility 2, 119
- vendor extension216
- vendor id119
- version number235
 - address lookup table 91, 154
 - call frame information 110, 157, 207
 - debug information 123, 188
 - line number information 95, 155, 203
 - name lookup table 90, 154
- virtuality attribute 29, 73, 77
 - encoding 149
- visibility attribute28
 - encoding 149
- void type *See* unspecified type entry
- volatile type entry68
- vtable element location attribute77
- with statement entry58