z/OS
2.5

DWARF/ELF Extensions Library Reference

SC14-7312-50

Note

Before using this information and the product it supports, read the information in “Notices” on page 203.

This edition applies to Version 2 Release 5 of z/OS® (5650-ZOS) and to all subsequent releases and modifications until otherwise indicated in new editions.

Last updated: 2021-09-30

Contents

About this document

This information is the reference for IBM extensions to the `libdwarf` and `libelf` libraries. It includes:

- Extensions to `libdwarf` consumer and producer APIs (Chapters 2 through 23)
- System-dependent APIs (Chapters 24-28)
- System-independent APIs (Chapters 29-30)
- Extensions to DWARF expression APIs (Chapter 31)
- Extensions to `libelf` utilities (Chapter 32-34)

This document discusses only these extensions, and does not provide a detailed explanation of standard DWARF and ELF APIs.

This document uses the following terminology:

ABI
Application binary interface. A standard interface by which an application gains access to system services, such as the operating-system kernel. The ABI defines the API plus the machine language for a central processing unit (CPU) family. The ABI ensures runtime compatibility between application programs and computer systems that comply with the standard.

API
Application programming interface. An interface that allows an application program that is written in a high-level language to use specific data or functions of the operating system or another program. An extension to a standard DWARF API can include:

- Extensions to standard DWARF files, objects, or operations
- Additional objects or operations

object
In object-oriented design or programming, a concrete realization (instance) of a class that consists of data and the operations associated with that data. An object contains the instance data that is defined by the class, but the class owns the operations that are associated with the data. Objects described in this document are generally a type definition or data structure, a container for a callback function prototype, or items that have been added to a DWARF file. See The DWARF industry-standard debugging information format and Example of a DWARF file.

operation
In object-oriented design or programming, a service that can be requested *at the boundary of an object*. Operations can modify an object or disclose information about an object.

Who should use this document

This document is intended for programmers who will be developing program analysis applications and debugging applications for the IBM® on the IBM z/OS operating system. The libraries provided by CDA allow applications to create or look for DWARF debugging information from ELF object files on the z/OS V1R10 operating system.

This document is a reference rather than a tutorial. It assumes that you have a working knowledge of the following items:

- The z/OS operating system
- The `libdwarf` APIs
- The `libelf` APIs
- The ELF ABI
- Writing debugging programs in C, C++ or COBOL on z/OS
- POSIX on z/OS

- The IBM Language Environment® on z/OS
- UNIX System Services shell on z/OS

A note about examples

Examples that illustrate the use of the `libelf`, `libdwarf`, and `libddpi` libraries are instructional examples, and do not attempt to minimize the run-time performance, conserve storage, or check for errors. The examples do not demonstrate all the uses of the libraries. Some examples are code fragments only, and cannot be compiled without additional code.

CDA and related publications

This section summarizes the content of the CDA publications and shows where to find related information in other publications.

Table 1. CDA, DWARF, ELF, and other related publications

Document title and number	Key sections/chapters in the document
z/OS Common Debug Architecture Library Reference	The reference for IBM's `libddpi` library. It includes: • General discussion of CDA • APIs with operations that access or modify information about stacks, processes, operating systems, machine state, storage, and formatting.
z/OS Common Debug Architecture User's Guide	The user's guide for the `libddpi` library. It includes: • Overview of the `libddpi` architecture. • Information on the order and purpose of calls to `libddpi` operations used to access DWARF information on behalf of model user applications. • Hints for using CDA with C/C++ source.
System V Application Binary Interface Standard	The Draft April 24, 2001 version of the ELF standard.
ELF Application Binary Interface Supplement	The Draft April 24, 2001 version of the ELF standard supplement.
DWARF Debugging Information Format, Version 3	The Draft 8 (November 19, 2001) version of the DWARF standard. This document is available on the web.
Consumer Library Interface to DWARF	The revision 1.48, March 31, 2002, version of the `libdwarf` consumer library.
Producer Library Interface to DWARF	The revision 1.18, January 10, 2002, version of the `libdwarf` producer library.
MIPS Extensions to DWARF Version 2.0	The revision 1.17, August 29, 2001, version of the MIPS extension to DWARF.

Table 1. CDA, DWARF, ELF, and other related publications (continued)

Document title and number	Key sections/chapters in the document
z/OS XL C/C++ User's Guide	Guidance information for: • z/OS C/C++ examples • Compiler options • Binder options and control statements • Specifying z/OS Language Environment run-time options • Compiling, IPA linking, binding, and running z/OS C/C++ programs • Utilities (Object Library, CXXFILT, DSECT Conversion, Code Set and Locale, ar and make, BPXBATCH, c89, xlc) • Diagnosing problems • Cataloged procedures and REXX EXECs supplied by IBM
z/OS XL C/C++ Programming Guide	Guidance information for: • Implementing programs that are written in C and C++ • Developing C and C++ programs to run under z/OS • Using XPLINK assembler in C and C++ applications • Debugging I/O processes • Using advanced coding techniques, such as threads and exception handlers • Optimizing code • Internationalizing applications
z/OS Enterprise COBOL Programming Guide, SC14-7382	Guidance information for: • Implementing programs that are written in COBOL • Developing COBOL programs to run under z/OS • z/OS COBOL examples • Compiler options • Compiling, linking, binding, and running z/OS COBOL programs • Diagnosing problems • Optimization and performance of COBOL programs • Compiler listings See Enterprise COBOL for z/OS documentation library (www.ibm.com/support/docview.wss?uid=swg27036733).

The following table lists the related publications for CDA, ELF, and DWARF. The table groups the publications according to the tasks they describe.

Table 2. Publications by task

Tasks	Documents
Coding programs	• *DWARF/ELF Extensions Library Reference, SC09-7655* • *z/OS Common Debug Architecture Library Reference, SC09-7654* • *z/OS Common Debug Architecture User's Guide, SC09-7653* • *DWARF Debugging Information Format* • *Consumer Library Interface to DWARF* • *Producer Library Interface to DWARF* • *MIPS Extensions to DWARF Version 2.0*
Compiling, binding, and running programs	• *z/OS XL C/C++ User's Guide, SC09-4767* • *z/OS XL C/C++ Programming Guide, SC09-4765* • *z/OS Enterprise COBOL Programming Guide, SC14-7382*
General discussion of CDA	• *z/OS Common Debug Architecture User's Guide, SC09-7653* • *z/OS Common Debug Architecture Library Reference, SC09-7654*
Environment and application APIs (objects and operations)	• *z/OS Common Debug Architecture Library Reference, SC09-7654*
A guide to using the libraries	• *z/OS Common Debug Architecture Library Reference, SC09-7654*
Examples of producer and consumer programs	• *z/OS Common Debug Architecture User's Guide, SC09-7653*

Softcopy documents

The following information describes where you can find softcopy documents.

The IBM z/OS Common Debug Architecture publications are supplied in PDF format and available for download at z/OS XL C/C++ documentation library (www.ibm.com/software/awdtools/czos/library)

To read a PDF file, use the Adobe Reader. If you do not have the Adobe Reader, you can download it (subject to Adobe license terms) from the Adobe web site at Adobe website (www.adobe.com).

Where to find more information

Please see *z/OS Information Roadmap* for an overview of the documentation associated with IBM z/OS.

How to send your comments

Your feedback is important in helping to provide accurate and high-quality information. If you have any comments about this document or the IBM documentation, send your comments by e-mail to: compinfo@cn.ibm.com

Be sure to include the name of the document, the part number of the document, the version of, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Chapter 1. About Common Debug Architecture

Common Debug Architecture (CDA) was introduced in z/OS V1R5 to provide a consistent format for debug information on z/OS. As such, it provides an opportunity to work towards a common debug information format across the various languages and operating systems that are supported on the IBM zSeries eServer™ platform. The product is implemented in the z/OS CDA libraries component of the z/OS Run-Time Library Extensions element of z/OS (V1R5 and higher).

CDA components are based on:

- The DWARF industry-standard debugging information format
- Executable and Linking Format (ELF) application binary interfaces (ABIs)

CDA-compliant applications can store DWARF debugging information in an ELF object file. However, the DWARF debugging information can be stored in any container. For example, in the case of the C/C++ compiler, the debug information is stored in a separate ELF object file, rather than the object file. In the case of the COBOL compiler, the debug information is stored in a GOFF object file, as well as the program object. In either approach, memory usage is minimized by avoiding the loading of debug information when the executable module is loaded into memory.

The DWARF industry-standard debugging information format

The DWARF 4 debugging format is an industry-standard format developed by the UNIX International Programming Languages Special Interest Group (SIG). It is designed to meet the symbolic, source-level debugging needs of different languages in a unified fashion by supplying language-independent debugging information. The debugging information format is open-ended, allowing for the addition of debugging information that accommodates new languages or debugger capabilities.

DWARF was developed by the UNIX International Programming Languages Special Interest Group (SIG).

The use of DWARF has two distinct advantages:

- It provides a stable and maintainable debug information format for all languages.
- It facilitates porting program analysis and debug applications to z/OS from other DWARF-compliant platforms.

Executable and Linking Format (ELF) application binary interfaces (ABIs)

Using a separate ELF object file to store debugging information enables the program analysis application to load specific information only as it is needed. With the z/OSXL C/C++ compiler, use the DEBUG option to create the separate ELF object file, which has a `*.dbg` extension.

Note: In this information, those ELF object files may be referred to as an ELF object file, an ELF object, or an ELF file. Such a file stores only DWARF debugging information.

GOFF program objects

Using a GOFF program object file enables the program analysis application to load specific information only as it is needed. With the Enterprise COBOL compiler, use the TEST option to create DWARF debugging information in the GOFF object file. The debugging information is stored in a NOLOAD class, and will not be loaded into memory when the program object is loaded into memory.

DWARF program information

The DWARF program information is block-structured for compatibility with the C/C++ (and other) language structures. DWARF does not duplicate information, such as the processor architecture, that is contained in the executable object.

The basic descriptive entity in a DWARF file is the *debugging information entry (DIE)*. DIEs can describe data types, variables, or functions, as well as other executable code blocks. A line table maps the executable instructions to the source that generated them.

The primary data types, built directly on the hardware, are the base types. DWARF base types provide the lowest level mapping between the simple data types and how they are implemented on the target machine's hardware. Other data types are constructed as collections or compositions of these base types.

A DWARF file is structured as follows:

- Each DWARF file is divided into debug sections.
- Each *debug section* provides information for a single compilation unit (CU) and contains one or more *DIE sections*.
- Each DIE section is identified with a unit header, which specifies the offset of the DIE section, and contains one or more DIEs.
- Each DIE has:
  - A tag that identifies the DIE. Each tag name has the DW_TAG prefix.
  - A section offset, which shows the relative position of the DIE within the DIE section.
  - A list of attributes, which fills in details and further describes the entity. Each attribute name has the DW_AT prefix.

    A DIE can have zero or more unique attributes. Each attribute must be unique to the DIE. In other words, a DIE cannot have two attributes of the same type but a DIE attribute type can be present in more than one DIE.
  - Zero or more children DIEs.

    Each descriptive entity in DWARF (except for the topmost entry which describes the source file) is contained within a parent entry and may contain child entities. If a DIE section contains multiple entities, all are siblings.
  - Nested-level indicators, which identify the parent/child relationship of the DIEs in the DIE section.

For detailed information about the DWARF format, see DWARF Debugging Standard (www.dwarfstd.org).

Example of a DWARF file

The example of a DWARF file is based on the output from the `dwarfdump` example program, and does not reflect an actual DWARF file that you might see in a normal program.

The example shows one debug section with one DIE section, which has two DIEs.

```
.debug_section_name                                 1
<unit header offset =0>unit_hdr_off:                2
<0><   11>      DW_TAG_DIE01                        3
                DW_AT_01          value00           4

<1><   20>      DW_TAG_DIE02                        5
                DW_AT_01          value01           6
                DW_AT_02          value02
                DW_AT_03          value03
```

Notes:

1. The name of each DWARF debug section starts with `.debug`.
2. The start of each DIE section is indicated by a line such as

   ```
   <unit header offset =0>unit_hdr_off:
   ```

The unit header offset indicates the relative location of the DIE sections within the DWARF debug section.

3. The start of the parent DIE is indicated by the line:`<0>< 11> DW_TAG_DIE01`, where:
   - `<0>` is the nested-level indicator that identifies the DIE as the parent of all DIEs in the DIE section with a nested-level indicator of `<1>`.
   - `<11>` is the section offset.
   - `DW_TAG_DIE01` is the DIE tag.
4. In the parent DIE, the attribute `DW_AT_01` is defined with `value00`. `DW_AT_01` is also used in `DW_TAG_DIE02`.
5. The start of the child DIE is indicated by the line:`<1>< 20> DW_TAG_DIE02`, where:
   - `<1>` is the nested-level indicator that identifies `DW_TAG_DIE01` as a child of `DW_TAG_DIE01`.
   - `<20>` is the section offset.
   - `DW_TAG_DIE02` is the DIE tag.
6. In the child DIE, the attribute `DW_AT_01` is defined with `value01`. `DW_AT_01` is also used in `DW_TAG_DIE01`.

IBM extensions to libdwarf

The `libdwarf` library contains interfaces to create and query DWARF debug objects.

`libdwarf` is a C library developed by Silicon Graphics Inc. (SGI). It provides:

- A consumer library interface to DWARF, which provides access to the DWARF debugging information
- A producer library interface to DWARF, which supports the creation of DWARF debugging information records
- Extensions to support SGI's MIPS processors

IBM has extended the `libdwarf` C/C++ library to support the z/OS operating system. The `libdwarf` library that is packaged with z/OS is available in 3 different forms:

- the 31-bit XPLINK version
- the 31-bit NOXPLINK version
- the 64-bit version

The CDA libraries provide a set of APIs to access DWARF debugging information. These APIs support the development of debuggers and other program analysis applications for z/OS.

IBM's extensions to `libdwarf` focus on:

- Improved speed and memory utilization
- z/OS XL C/C++ Support for the languages
- Enterprise COBOL support
- z/OS future support for languages such as FORTRAN, HLASM, PL/I,

Changes to DWARF/ELF library extensions

This section provides a summary of changes that are shipped with the DWARF/ELF libraries.

the DW_FRAME_390_REG_type data structure has been updated to add the following vector registers:

- DW_FRAME_390_vr0
- DW_FRAME_390_vr1
- DW_FRAME_390_vr2
- DW_FRAME_390_vr3
- DW_FRAME_390_vr4

- DW_FRAME_390_vr5
- DW_FRAME_390_vr6
- DW_FRAME_390_vr7
- DW_FRAME_390_vr8
- DW_FRAME_390_vr9
- DW_FRAME_390_vr10
- DW_FRAME_390_vr11
- DW_FRAME_390_vr12
- DW_FRAME_390_vr13
- DW_FRAME_390_vr14
- DW_FRAME_390_vr15
- DW_FRAME_390_vr16
- DW_FRAME_390_vr18
- DW_FRAME_390_vr20
- DW_FRAME_390_vr22
- DW_FRAME_390_vr17
- DW_FRAME_390_vr19
- DW_FRAME_390_vr21
- DW_FRAME_390_vr23
- DW_FRAME_390_vr24
- DW_FRAME_390_vr26
- DW_FRAME_390_vr28
- DW_FRAME_390_vr30
- DW_FRAME_390_vr25
- DW_FRAME_390_vr27
- DW_FRAME_390_vr29
- DW_FRAME_390_vr31

Chapter 2. Debugging Information Entry (DIE) extensions

This chapter describes IBM extensions to information within the `.debug_info` section.

Program scope entries

This section describes debugging information entries that relate to different levels of program scope, including compilation, module, subprogram, and so on.

Normal and partial compilation unit entries

A normal compilation unit is represented by a debugging information entry with the tag `DW_TAG_compile_unit` (known as CU DIE hence forth). Each CU DIE may have a `DW_AT_stmt_list` attribute whose value is a section offset to the line number information for this compilation unit. A separate line number table is generated for each source view, and the line number table that is associated with the CU DIE is the default source view (user source).

For each additional source view (for example, Assembly View), there is a `DW_TAG_IBM_src_view` DIE. The parent of this DIE is the CU DIE. The `DW_TAG_IBM_src_view` DIE has the following attributes:

- A `DW_AT_name` attribute, whose value is a null-terminated string containing the name of the source view.
- A `DW_AT_stmt_list` attribute, whose value is a section offset to the line number information for this source view.
- A `DW_AT_IBM_src_file` attribute, whose value is a DIE section offset to the `.debug_srcfiles` section. The referenced source file DIE contains additional information about the primary source file within the source view.

A CU DIE may have the following attributes:

- A `DW_AT_IBM_alt_srcview` attribute, which points to the `DW_TAG_IBM_src_view` DIE.
- A `DW_AT_IBM_branch_flags` attribute, which is a flag whose presence indicates that the program's branch points are identified by the bits in `sysattr_flag` on the line number entries.
- A `DW_AT_IBM_charset` attribute, which is a string representing the codeset that is used by the compiler to interpret the identifier names within this compilation unit.
- A `DW_AT_IBM_has_baselist` attribute, which is a flag whose presence indicates that at least one base location entry exists in the compilation unit.
- A `DW_AT_IBM_has_xref` attribute, which is a flag whose presence indicates that the producer supports the generation of cross-reference information. For example, the producer can produce `.debug_xref` section if cross-reference information is available.
- A `DW_AT_IBM_line_reordered` attribute, which is a flag whose presence indicates that the execution order of the statements within the line number program may not match the flow of the original source program. (This only applies to those statements without synchronization flag).
- A `DW_AT_IBM_set_unreliable` attribute, which is a flag whose presence indicates that when a debugger is stopped on an executable statement, it cannot reliably modify the content of variable and have the new value reflected for the rest of the execution.
- A `DW_AT_IBM_sync_point` attribute, which is a flag indicating that when a debugger is stopped on an executable statement, it cannot reliably modify the content of a variable and have the new value reflected for the rest of the execution.
- A `DW_AT_identifier_case` attribute, whose integer constant value is a code describing the treatment of identifiers within this compilation unit.

- A DW_AT_linkage_name attribute, whose value is a null-terminated string describing the program name that is associated with the compilation unit. For COBOL, this contains the program-id name that is specified in the source program.
- 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 the following DWARF samples:

DWARF sample: .debug_info

```
$1: DW_TAG_compile_unit
      DW_AT_stmt_list (...)
      DW_AT_low_pc (...)
      DW_AT_high_pc (...)
      DW_AT_IBM_alt_srcview (...)
$2: DW_TAG_IBM_src_view
      DW_AT_name (Assembly View)
      DW_AT_stmt_list (...)
      DW_AT_IBM_src_file ($5)
```

DWARF sample: .debug_srcfiles

```
$5: DW_TAG_IBM_src_file
      DW_AT_name  (Assembly View)
      DW_AT_IBM_src_type (DW_SFT_compiler_generated)
      DW_AT_IBM_src_text (...)
      DW_AT_IBM_md5 (0123456789abcdef0123456789abcdef)
      DW_AT_IBM_src_attr (...)
```

Byte and bit entries

Many debugging information entries allow either a DW_AT_byte_size attribute or a DW_AT_bit_size attribute, whose value specifies an amount of storage. The value of the DW_AT_byte_size attribute is interpreted in bytes and the value of the DW_AT_bit_size attribute is interpreted in bits.

The value of the attribute is determined based on the class as follows:

- For a constant, the value of the constant is the value of the attribute.
- For a reference, the value is a reference to another entity which specifies the value of the attribute.
- For an exprloc, the value is interpreted as a DWARF expression. Evaluation of the expression yields the value of the attribute.

Subroutine and entry point entries

A subroutine or entry point entry may have a DW_AT_frame_base attribute, whose value is a location description that computes the frame base for the subroutine or entry point. If the location description is a simple register location description, the given register contains the frame base address. If the location description is a DWARF expression, the result of evaluating that expression is the frame base address. Finally, for a location list, this interpretation applies to each location description contained in the list of location list entries.

For COBOL, the DW_AT_frame_base attribute provides the base location for all the local storage within the subprogram.

If a subprogram or entry point is nested, it has a DW_AT_static_link attribute, whose value is a location description that computes the frame base of the subprogram that immediately encloses the subprogram or entry point. To resolve an up-level reference to a variable, a debugger must use the nesting structure of DWARF to determine which subprogram is the lexical parent and the DW_AT_static_link value to identify the appropriate frame base of the parent subprogram.

Source view entries

For each additional source view (for example, Assembly View), there is one DW_TAG_IBM_src_view DIE. The parent of this DIE is the CU DIE. It has the following attributes:

- A DW_AT_name attribute, whose value is a null-terminated string containing the name of the source view.
- A DW_AT_stmt_list attribute, whose value is a section offset to the line number information for this source view.
- A DW_AT_IBM_src_file attribute, whose value is a DIE section offset to the .debug_srcfiles section. The referenced source file DIE (DW_TAG_IBM_src_file or DW_TAG_IBM_src_filelist) contains information about the source file(s) referenced within the line number program.

See the following sample source view DWARF entries:

```
.debug_info
$1: DW_TAG_compile_unit
    DW_AT_stmt_list (...)
    DW_AT_IBM_src_file (...)
    DW_AT_low_pc (...)
    DW_AT_high_pc (...)
$2: DW_TAG_IBM_src_view
    DW_AT_name (Assembly View)
    DW_AT_stmt_list (...)
    DW_AT_IBM_src_file ($5)

.debug_srcfiles
$5: DW_TAG_IBM_src_file
      DW_AT_name  (Assembly View)
      DW_AT_IBM_src_type (DW_SFT_compiler_generated)
      DW_AT_IBM_src_text (...)
      DW_AT_IBM_md5 (0123456789abcdef0123456789abcdef)
      DW_AT_IBM_src_attr (...)
```

Object oriented COBOL

COBOL has the notion of class-id, which provides a way for the compiler to create a Java™ class with the specified name. Within this class, class methods and data can be declared.

A COBOL class is represented by a debugging information entry with the tag DW_TAG_namespace. It has a DW_AT_name attribute, whose value is a null-terminated string containing the class name as it appears in the source program. The debugging information entries for the class methods and data will be children of the DW_TAG_namespace.

Data object and object list entries

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

Data object entries

Some languages (such as COBOL) have the concept of grouping objects into different sections. The section grouping specifies the section which the object belongs to.

The section grouping is represented by a DW_AT_IBM_section_grouping attribute, whose value is a constant:

DW_SG_cobol_working	0x0	COBOL WORKING-STORAGE SECTION
DW_SG_cobol_linkage	0x1	COBOL LINKAGE SECTION
DW_SG_cobol_file	0x2	COBOL FILE SECTION

DW_SG_cobol_local	0x3	COBOL LOCAL-STORAGE SECTION
DW_SG_cobol_special_register	0x4	COBOL Special Registers

See the following COBOL snippet:

```
WORKING-STORAGE SECTION.
01 UBIN4 PIC 9(4) USAGE BINARY.

LOCAL-STORAGE SECTION.
01 SBIN0_1 PIC SV9 USAGE BINARY.
```

See the following DWARF sample:

```
$1: DW_TAG_variable
      DW_AT_name (UBIN4)
      DW_AT_type (PIC 9(4))
      DW_AT_IBM_section_grouping (DW_SG_cobol_working)
      DW_AT_location (...)
$2: DW_TAG_variable
      DW_AT_name (SBIN0_1)
      DW_AT_type (PIC SV9)
      DW_AT_IBM_section_grouping (DW_SG_cobol_local)
      DW_AT_location (...)
```

Referencing coordinates

Any debugging information entry representing an object, module, or subprogram may have a DW_AT_IBM_xref_coord attribute whose value is a data block form. This can be used to indicate all the occurrence of a variable in the program source.

The value of the DW_AT_IBM_xref_coord attribute contains at least one pair of unsigned LEB128 numbers representing the source line number and source column number at which the first character of the identifier of the referencing object appears. The source column number 0 indicates that no column has been specified. To conserve space, the source line numbers are sorted in ascending order.

Only the first pair of unsigned LEB128 contains the actual source line number and source column number. In the subsequent pairs, the first number contains the delta source line number, that is the actual source line number minus the source line number or the previous entry. The column number for each pair contains the actual source column number.

For example, in the code sample below:

```
               1         2         3
     01234567890123456789012345678901234567890
     ----------|---------|---------|----------
0149:            Display s15a
0150:            Compute s30 = s15a * s15a
```

the variable s15a appears in three places at source coordinates: 149,20;150,26 and 150,33. These 3 pairs of values are encoded as:

```
149,20;1,26;0,33
```

Base location entries

Some language may group the location of data objects under a common location anchor. For example, in COBOL, all the local storage items are grouped together at a specific storage location with a predefined length.

A base location list is represented by a debugging information entry with the tag DW_TAG_IBM_location_baselist. The base location list is only applicable within the address range defined by its parent debugging information entry. For example, if the parent of the debugging information

entry is the compilation unit DIE, the base location list is applicable when the current program counter is within the address range of the compilation unit.

Each base location item that is a part of the base location list is represented by a debugging information entry with the tag `DW_TAG_IBM_location_base`. Each such entry is a child of the base location list entry. Each base location item entry contains a `DW_AT_location` attribute, whose value is a location description, describing how to find the starting address of the base location item. Each base location item entry may contain a `DW_AT_byte_size` attribute whose value is the length of data in bytes described by this base location item. The value of the attribute is determined as described in "Byte and bit entries" on page 6

Each base location item entry may contain a `DW_AT_IBM_location_type` attribute whose value describes the data referenced by the base location item. The value is a constant drawn from the set of following codes:

DWARF location type name	Value	Description
DW_LT_cobol_file	0	COBOL file data
DW_LT_cobol_linkage	1	COBOL linkage data
DW_LT_cobol_external	2	COBOL external data
DW_LT_cobol_oo	3	COBOL object oriented data
DW_LT_cobol_xml	4	COBOL XML data
DW_LT_rent24	5	24-bit reentrant data
DW_LT_rent32	6	32-bit reentrant data
DW_LT_norent32	7	32-bit non-reentrant data

Type entries

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

Base type entries

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_encoding` attribute describing how the base type is encoded and is to be interpreted. The value of this attribute is an integer constant. IBM extensions are introduced to describe the following data types:

DW_AT_encoding name	Value	Description
DW_ATE_IBM_complex_float_hex	0xde	IBM hex complex floating point
DW_ATE_IBM_float_hex	0xdf	IBM hex floating point
DW_ATE_IBM_imaginary_float_hex	0xe0	IBM hex imaginary floating point
DW_ATE_IBM_edited_national	0xe5	COBOL national numeric edited data type
DW_ATE_IBM_edited_DBCS	0xe6	COBOL DBCS edited data type
DW_ATE_IBM_external_float	0xe7	COBOL external floating point data type
DW_ATE_IBM_external_float_national	0xe8	COBOL national external floating point data type
DW_ATE_IBM_string_national	0xe9	COBOL national alphanumeric data type
DW_ATE_IBM_string_DBCS	0xea	COBOL DBCS alphanumeric data type

DW_AT_encoding name	Value	Description
DW_ATE_IBM_numeric_string_national	0xeb	COBOL national numeric data type
DW_ATE_IBM_index_name	0xec	COBOL index name
DW_ATE_IBM_index_data_item	0xed	COBOL index data item

DWARF standard encoding is used for the following data types:

DW_ATE_packed_decimal	0x0a	COBOL unsigned or signed packed decimal (COMP-3)
DW_ATE_numeric_string	0x0b	COBOL zoned decimal (unsigned, sign trailing included, sign trailing separate, sign leading included, or sign leading separate)
DW_ATE_edited	0x0c	COBOL alphanumeric edited, COBOL numeric edited
DW_ATE_signed_fixed	0x0d	COBOL signed COMP-4 or COMP-5
DW_ATE_unsigned_fixed	0x0e	COBOL unsigned COMP-4 or COMP-5
DW_ATE_UTF	0x10	COBOL UTF-8 string type

In COBOL, a base type entry may have a DW_AT_picture_string attribute whose value is a null-terminated string containing the picture string as specified in the source code.

A base type entry has either a DW_AT_byte_size attribute or a DW_AT_bit_size attribute whose integer constant value is the amount of storage needed to hold a value of the type.

A packed decimal type (for example, DW_ATE_packed_decimal) may have a DW_AT_decimal_sign attribute, whose value is an integer constant that conveys the representation of the sign of the decimal type. The only allowable value is DW_DS_unsigned. Absence of the attribute indicates that there is a sign in the encoding.

A zoned decimal type (for example, DW_ATE_numeric_string and DW_ATE_IBM_numeric_string_national) may have a DW_AT_decimal_sign attribute, whose value is an integer constant that conveys the representation of the sign of the decimal type. 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.

A decimal sign attribute has the following values:

DW_DS_unsigned	0x01	unsigned
DW_DS_leading_overpunch	0x02	Sign is encoded in the most significant digit in a target-dependent manner.
DW_DS_trailing_overpunch	0x03	Sign is encoded in the least significant digit in a target-dependent manner.
DW_DS_leading_separate	0x04	Sign is a + or - character to the left of the most significant digit.
DW_DS_trailing_separate	0x05	Sign is a + or - character to the right of the least significant digit.

In COBOL, a native binary number type (for example, DW_ATE_signed_fixed and DW_ATE_unsigned_fixed) has a DW_AT_IBM_native_binary attribute, which is a flag. This attribute indicates that the data item is represented in storage as native binary data.

A fixed-point scaled integer base type (for example, DW_ATE_numeric_string, DW_ATE_signed_fixed, DW_ATE_unsigned_fixed, DW_ATE_packed_decimal, and

DW_ATE_numeric_string_national) or a COBOL numeric edited type (for example, DW_ATE_edited and DW_ATE_IBM_edited_national) has the following attributes:

- A DW_AT_digit_count attribute, whose value is an integer constant that represents the number of digits in an instance of the type.
- A DW_AT_decimal_scale attribute, whose value is an integer constant 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.

An alphanumeric base type (for example, DW_ATE_IBM_string_national and DW_ATE_IBM_string_DBCS) may have a DW_AT_IBM_justify attribute, which is a flag. This attribute indicates whether the object is justified to the right.

An UTF-8 string type (DW_ATE_UTF) might have a DW_AT_IBM_string_length attribute, whose value is an integer constant that represents the number of characters of the string.

A COBOL index name (for example, DW_ATE_IBM_index_name), has a DW_AT_byte_stride attribute, whose value is the size of each table entry.

See the following DWARF sample:

```
* pic ABBA(5).
DW_TAG_base_type
  DW_AT_encoding (DW_ATE_edited)
  DW_AT_picture_string (ABBA(5))
  DW_AT_byte_size (8)

* pic S999V999 SIGN TRAILING.
DW_TAG_base_type
  DW_AT_encoding (DW_ATE_numeric_string)
  DW_AT_picture_string (S999V999)
  DW_AT_byte_size (6)
  DW_AT_decimal_sign (DW_DS_trailing_overpunch)
  DW_AT_digit_count (6)
  DW_AT_decimal_scale (-3)

* pic S999V99 USAGE BINARY.
DW_TAG_base_type
  DW_AT_encoding (DW_ATE_signed_fixed)
  DW_AT_picture_string (S999V99)
  DW_AT_byte_size (4)
  DW_AT_digit_count (5)
  DW_AT_decimal_scale (-2)

* pic S9(3)V99 PACKED-DECIMAL.
DW_TAG_base_type
  DW_AT_encoding (DW_ATE_packed_decimal)
  DW_AT_picture_string (S9(3)V99)
  DW_AT_byte_size (3)
  DW_AT_digit_count (5)
  DW_AT_decimal_scale (-2)

* pic 999PP COMP-3.
DW_TAG_base_type
  DW_AT_encoding (DW_ATE_packed_decimal)
  DW_AT_decimal_sign (DW_DS_unsigned)
  DW_AT_picture_string (999PP)
  DW_AT_byte_size (2)
  DW_AT_digit_count (3)
  DW_AT_decimal_scale (2)

* pic +Z,ZZ9.99.
DW_TAG_base_type
  DW_AT_encoding (DW_ATE_edited)
  DW_AT_picture_string (+Z,ZZ9.99)
  DW_AT_byte_size (9)
  DW_AT_digit_count (6)
  DW_AT_decimal_scale (-2)

* pic 9999/99 USAGE NATIONAL.
DW_TAG_base_type
  DW_AT_encoding (DW_ATE_IBM_edited_national)
```

```
  DW_AT_picture_string (9999/99)
  DW_AT_byte_size (14)
  DW_AT_digit_count (6)
  DW_AT_decimal_scale (0)

* pic N(4) USAGE NATIONAL.
DW_TAG_base_type
  DW_AT_encoding (DW_ATE_IBM_string_national)
  DW_AT_picture_string (N(4))
  DW_AT_byte_size (8)

* pic NNBBNN USAGE NATIONAL.
DW_TAG_base_type
  DW_AT_encoding (DW_ATE_IBM_edited_national)
  DW_AT_picture_string (NNBBNN)
  DW_AT_byte_size (12)

* pic UUUU.
DW_TAG_base_type
  DW_AT_encoding (DW_ATE_UTF)
  DW_AT_picture_string (UUUU)
  DW_AT_byte_size (16)
  DW_AT_IBM_string_length (4)

* 01  year-accum.
*   02  month-entry occurs 12 indexed by IDXNAME.
*     03  STABS  USAGE IDXITEM.
DW_TAG_base_type     * IDXNAME
  DW_AT_encoding (DW_ATE_IBM_index_name)
  DW_AT_byte_size (4)
  DW_AT_byte_stride (4)
DW_TAG_base_type     * IDXITEM
  DW_AT_encoding (DW_ATE_IBM_index_data_item)
  DW_AT_byte_size (4)
```

Modified type entries

A modified type entry describing a COBOL function pointer is represented by a debugging information entry with the tag DW_TAG_IBM_funcptr_type. It may have a DW_AT_address_class attribute, whose value is an integer, to describe how objects having the given pointer type ought to be dereferenced. It has a DW_AT_type attribute, whose value is a reference to a debugging information entry describing a base type.

A modified type entry describing a COBOL procedure pointer is represented by a debugging information entry with the tag DW_TAG_IBM_procptr_type. It may have a DW_AT_address_class attribute, whose value is an integer, to describe how objects having the given pointer type ought to be dereferenced. It has a DW_AT_type attribute, whose value is a reference to a debugging information entry describing a base type.

A modified type entry describing a COBOL object reference is represented by a debugging information entry with the tag DW_TAG_IBM_objref_type. It may have a DW_AT_address_class attribute, whose value is an integer, to describe how objects having the given pointer type ought to be dereferenced. It has a DW_AT_type attribute, whose value is a reference to a debugging information entry describing a base type.

Structure, union, class and interface 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.

In COBOL, a group is by default alphanumeric. When a GROUP-USAGE NATIONAL clause is declared for a group item, then the corresponding structure type entry has a DW_AT_type attribute, whose value is a reference to the debugging information entry describing the national type. If the attribute is absent, the group is by default alphanumeric.

See the following COBOL snippet:

```
1 GRP2 GROUP-USAGE NATIONAL.
       3 DUPU pic N(20).
```

See the following DWARF sample:

```
$1: DW_TAG_base_type
      DW_AT_encoding (DW_ATE_IBM_string_national)
$2: DW_TAG_structure_type
      DW_AT_type ($1)
$3: DW_TAG_member
      DW_AT_name (DUPU)
$4: DW_TAG_variable
      DW_AT_name (GRP2)
      DW_AT_type ($2)
```

Some languages (such as COBOL) have the concept of assigning level number to a structure and its members. The level number defines the parent/child relationship for the structure members.

A debugging information entry that represents a program variable (for example, `DW_TAG_variable`) or a data member entry (for example, `DW_TAG_member`) may have a `DW_AT_IBM_level_number` attribute, whose value is an integer constant.

See the following COBOL snippet:

```
01   EMPLOYEE-RECORD.
     05  EMPLOYEE-NAME.
         10  FIRST     PICTURE  X(10).
         10  LAST      PICTURE  X(10).
     05  EMPLOYEE-ADDRESS.
         10  STREET    PICTURE  X(10).
         10  CITY      PICTURE  X(10).
```

See the following DWARF sample:

```
$01: DW_TAG_structure_type
       DW_AT_name (EMPLOYEE-NAME)
$02: DW_TAG_member
       DW_AT_name (FIRST)
       DW_AT_IBM_level_number (10)
$03: DW_TAG_member
       DW_AT_name (LAST)
       DW_AT_IBM_level_number (10)
$05: DW_TAG_structure_type
       DW_AT_name (EMPLOYEE-ADDRESS)
$06: DW_TAG_member
       DW_AT_name (STREET)
       DW_AT_IBM_level_number (10)
$07: DW_TAG_member
       DW_AT_name (CITY)
       DW_AT_IBM_level_number (10)
$10: DW_TAG_structure_type
       DW_AT_name (EMPLOYEE-RECORD)
$11: DW_TAG_member
       DW_AT_name (EMPLOYEE-NAME)
       DW_AT_type ($01)
       DW_AT_IBM_level_number (5)
$12: DW_TAG_member
       DW_AT_name (EMPLOYEE-ADDRESS)
       DW_AT_type ($05)
       DW_AT_IBM_level_number (5)
$20: DW_TAG_variable
       DW_AT_name (EMPLOYEE-RECORD)
       DW_AT_IBM_level_number (1)
       DW_AT_type ($10)
```

String type entries

A `string` is a sequence of characters that have specific semantics and operations that separate them from arrays of characters. A string type is represented by a debugging information entry with the tag `DW_TAG_string_type`. In COBOL, this corresponds to an alphabetic or alphanumeric type.

A string type may have a `DW_AT_byte_size` attribute whose value is the amount of storage needed to hold a value of the string type.

In COBOL, a string type entry has a DW_AT_picture_string attribute whose value is a null-terminated string containing the picture string as specified in the source code.

In COBOL, a string type entry may have a DW_AT_IBM_justify attribute, which is a flag. This attribute indicates whether the object is justified to the right.

In COBOL, a string type entry may have a DW_AT_IBM_is_alphabetic attribute, which is a flag. This attribute indicates that the object is an alphabetic type. Absence of this attribute indicates that the object is an alphanumeric type.

See the following DWARF sample:

```
* pic A(10) JUST RIGHT.
DW_TAG_string_type
  DW_AT_picture_string (A(10))
  DW_AT_IBM_justify (yes)
  DW_AT_IBM_is_alphabetic (yes)
  DW_AT_byte_size (10)
```

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 or value ranges. Semantically, the condition is true if the value of the conditional variable matches any of the described constants, and the condition is false otherwise.

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

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

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

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

See the following COBOL snippet:

```
1  ALPHA    PIC X(10).
88 TESTALPHA VALUE 'TOM', 'FRED', 'A' thru 'Z'.
```

See the following DWARF sample:

```
$01: DW_TAG_string_type
       DW_AT_picture_string (X(1))
       DW_AT_byte_size (1)
$02: DW_TAG_string_type
       DW_AT_picture_string (X(3))
       DW_AT_byte_size (3)
$03: DW_TAG_string_type
       DW_AT_picture_string (X(4))
       DW_AT_byte_size (4)
$10: DW_TAG_variable
       DW_AT_name (ALPHA)
$11: DW_TAG_condition
```

```
       DW_AT_name (TESTALPHA)
$12: DW_TAG_constant
       DW_AT_const_value (c1)       ! 'A'
       DW_AT_type ($01)
$13: DW_TAG_constant
       DW_AT_const_value (e9)       ! 'Z'
       DW_AT_type ($01)
$14: DW_TAG_subrange_type
       DW_AT_lower_bound ($12)
       DW_AT_upper_bound ($13)
$15: DW_TAG_constant
       DW_AT_const_value (c6d9c5c4) ! 'FRED'
       DW_AT_type ($03)
$16: DW_TAG_constant
       DW_AT_const_value (e3d6d4)   ! 'TOM'
       DW_AT_type ($02)
```

File description entries

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

The COBOL file name debugging information entry has a DW_AT_type attribute referencing a file description entry type with the tag DW_TAG_file_type. If the file description entry type describes a COBOL file description, the file description entry type has a DW_AT_name attribute, whose value is a string FD. If the file description entry type describes a COBOL sort file description entry, it has a DW_AT_name attribute, whose value is a string SD.

The COBOL file name debugging information entry may have a DW_AT_location attribute, whose value is a location description. The result of evaluating this description yields the FCB of the file description entry.

If a GLOBAL clause is specified on the COBOL file name, then the file name debugging information entry has a DW_AT_visibility attribute, whose value is DW_VIS_exported.

Each top level record debugging information entries is represented by a debugging information entry with the tag DW_TAG_variable. It may have attributes similar to those debugging information entries for top level structure variable. In addition, it has a DW_AT_IBM_owner attribute, whose value is a reference to the owning COBOL file name debugging information entry.

See the following COBOL snippet:

```
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT RUNDATA
       ASSIGN TO SYSIN-S-FILE1DD
       ORGANIZATION IS SEQUENTIAL
       ACCESS MODE IS SEQUENTIAL
       FILE STATUS IS RUNDATA-FS.
    SELECT SORT-FILE
       ASSIGN TO SORTFILE.
DATA DIVISION.
FILE SECTION.
FD  RUNDATA DATA RECORD WEEK LABEL RECORDS OMITTED
                BLOCK CONTAINS 0 RECORDS.
01  WEEK.
    03  MONTH  PICTURE 99.
01  SALARY     PIC 9(4)V9(2).
SD  SORT-FILE.
01  SORT-REC.
    02 SD-FN   PICTURE X(20).
```

See the following DWARF sample:

```
$1: DW_TAG_file_type
      DW_AT_name (FD)
$2: DW_TAG_file_type
      DW_AT_name (SD)

$5: DW_TAG_variable
      DW_AT_name (RUNDATA)
```

```
      DW_AT_type ($1)
      DW_AT_location (location of FCB)
$6: DW_TAG_variable
      DW_AT_name (WEEK)
      DW_AT_type (struct for WEEK)
      DW_AT_IBM_owner ($5)    ! indicator for children of RUNDATA
      DW_AT_IBM_level_number (1)
      DW_AT_location (...)
$7: DW_TAG_variable
      DW_AT_name (SALARY)
      DW_AT_type (PIC 9(4)V9(2))
      DW_AT_IBM_owner ($5)    ! indicator for children of RUNDATA
      DW_AT_IBM_level_number (1)
      DW_AT_location (...)

$8: DW_TAG_variable
      DW_AT_name (SORT-FILE)
      DW_AT_type ($2)         ! no FCB
$9: DW_TAG_variable
      DW_AT_name (SORT-REC)
      DW_AT_type (struct for SORT-REC)
      DW_AT_IBM_owner ($8)    ! indicator for children of RUNDATA
      DW_AT_IBM_level_number (1)
      DW_AT_location (...)
```

Bound checking information for type entries

Some languages (such as COBOL) have well defined upper and lower storage limits for objects whose storage is dynamically determined at runtime. Knowing the storage limits allows the debugger to perform bounds checking when examining data objects with these types.

A type entry whose storage is not known during compilation may have the following attributes:

- A DW_AT_IBM_max_upper_bound attribute whose integer constant value specifies the upper bound associated with the upper storage limit that can be used to hold a data object of this type.
  The value of the DW_AT_IBM_max_upper_bound attribute together with DW_AT_byte_stride or DW_AT_bit_stride can be used to calculate the upper storage limit for a data object of this type.
- A DW_AT_IBM_min_upper_bound attribute whose integer constant value specifies the upper bound associated with the lower storage limit that can be used to hold a data object of this type.
  The value of the DW_AT_IBM_min_upper_bound attribute together with DW_AT_byte_stride or DW_AT_bit_stride can be used to calculate the lower storage limit for a data object of this type.

For data members whose offsets are calculated at runtime, the offset calculation (specified on DW_AT_data_location) may fail if certain precondition is not met. For example, the offset calculation may rely on some other variable to be within range before the calculation can yield correct result. A data member with this characteristic may have a DW_AT_IBM_valid_expr attribute, whose value is a DWARF expression. If the result of the evaluation is zero, then the precondition for evaluating the offset has not been met.

For data types whose lengths are calculated at runtime, the length calculation (specified on DW_AT_byte_size) may fail if certain precondition is not met. For example, the length calculation may rely on some other variable to be within range before the calculation can yield correct result. A data type with this characteristic may have a DW_AT_IBM_valid_expr attribute, whose value is a DWARF expression. If the result of the evaluation is zero, then the precondition for evaluating the length has not been met.

For array types whose strides are calculated at runtime, the stride calculation (specified on DW_AT_byte_stride) may fail if certain precondition is not met. For example, the stride calculation may rely on some other variable to be within range before the calculation can yield correct result. A data type with this characteristic may have a DW_AT_IBM_valid_expr attribute, whose value is a DWARF expression. If the result of the evaluation is zero, then the precondition for evaluating the stride has not been met.

See the following COBOL snippet:

```
01  CSSS7.
    05 CSSS7-C PIC X(4) VALUE "CCCC".
```

```
    05 CS7-F.
       10 CS7-H  PIC 9(2) OCCURS 10 TIMES
                    DEPENDING ON OBJ-7C INDEXED BY CS7-IX3.
    05  INCREMENT PIC 99.
```

See the following DWARF sample:

```
$01: DW_TAG_variable
       DW_AT_name (CSSS7)
       DW_AT_type ($02)
       DW_AT_IBM_level_number (1)
       DW_AT_location (...)

$02: DW_TAG_structure_type
       DW_AT_byte_size (DW_OP_call_ref ... DW_OP_lit10 DW_OP_plus)
       DW_AT_IBM_valid_expr (DW_OP_call_ref $12)
$03: DW_TAG_member
       DW_AT_name (CSSS7-C)
       DW_AT_type (...)  *PIC X(4)
       DW_AT_data_member_location (DW_OP_plus_uconst 0)
       DW_AT_IBM_level_number (5)
$04: DW_TAG_member
       DW_AT_name (CS7-F)
       DW_AT_type ($06)
       DW_AT_data_member_location (DW_OP_plus_uconst 4)
       DW_AT_IBM_level_number (5)
$05: DW_TAG_member
       DW_AT_name (INCREMENT)
       DW_AT_type (...)  *PIC 99
       DW_AT_data_member_location (DW_OP_call_ref ... DW_OP_plus)
       DW_AT_IBM_valid_expr (DW_OP_call_ref $12)
       DW_AT_IBM_level_number (5)

$06: DW_TAG_structure_type
       DW_AT_byte_size (DW_OP_call_ref ...)
       DW_AT_IBM_valid_expr (DW_OP_call_ref $11)
$07: DW_TAG_member
       DW_AT_name (CS7-H)
       DW_AT_type ($08)
       DW_AT_data_member_location (DW_OP_plus_uconst 0)
       DW_AT_IBM_level_number (10)

$08: DW_TAG_array_type
       DW_AT_type (...)
       DW_AT_byte_size (DW_OP_call_ref ...)
       DW_AT_IBM_valid_expr (DW_OP_call_ref $10)
$09: DW_TAG_subrange_type
       DW_AT_lower_bound (1)
       DW_AT_upper_bound (DW_OP_call_ref ...)
       DW_AT_IBM_max_upper_bound (10) * Maximum upper bound

// Bound checking for array type ($08)
$10: DW_TAG_dwarf_procedure
       DW_AT_location (DW_OP_call_ref <upper_bound expr>
              DW_OP_dup 1 DW_OP_ge 10 DW_OP_le DW_OP_and)

// Bound checking for struct type ($06)
$11: DW_TAG_dwarf_procedure
       DW_AT_location (DW_OP_call_ref $10)

// Bound checking for struct type ($02)
$12: DW_TAG_dwarf_procedure
       DW_AT_location (DW_OP_call_ref $11)
```

Chapter 3. Consumer APIs for standard DWARF sections

These are IBM's extended consumer operation and the macros that it uses to access the standard DWARF sections.

Error object consumer operations

This section contains a list of APIs for accessing information within a DWARF error object.

When an error occurs and an error object is passed to the API, the error object will contain a value indicating the type of that error.

If an error object is not passed to the API, that is NULL is the last parameter, the API creates an error object and calls the error handler routine that has been specified in the `libdwarf` initialization routine.

If an error object is not passed to the API and you have not specified an error handler routine when initializing the`libdwarf` consumer object, the API will not complete.

Error handling macros

This topic is a list of error values that are represented in a returned error object.

DW_DLE_INVALID_GOFF_RELOC
Value is 1. The Dwarf_Goff_Reloc object is NULL or not valid.

DW_DLE_ID
Value is 6. The register number specified is out of range.

DW_DLE_IA
Value is 9. The Dwarf_Debug or Dwarf_P_Debug object is corrupted and there is an eye-catcher mismatch.

DW_DLE_FNO
Value is 12. Unable to open file for processing.

DW_DLE_FNR
Value is 13. The file name specified is not valid.

DW_DLE_NOB
Value is 15. The input file format is not recognized.

DW_DLE_BADBITC
Value is 22. Invalid/Incompatible address size detected.

DW_DLE_DBG_ALLOC
Value is 23. Unable to malloc a Dwarf_Debug/Dwarf_P_Debug object.

DW_DLE_FSTAT_ERROR
Value is 24. fstat() failed.

DW_DLE_FSTAT_MODE_ERROR
Value is 25. The file mode bits do not indicate that the file being opened is a normal file.

DW_DLE_INIT_ACCESS_WRONG
Value is 26. The file access mode specified is not valid.

DW_DLE_ELF_BEGIN_ERROR
Value is 27. A call to elf_begin() failed.

DW_DLE_ELF_GETEHDR_ERROR
Value is 28. A call to elf32_getehdr() or elf64_getehdr() failed.

DW_DLE_ELF_GETSHDR_ERROR
Value is 29. A call to elf32_getshdr() or elf64_getshdr() failed.

DW_DLE_ELF_STRPTR_ERROR
Value is 30. A call to elf_strptr() failed trying to get a section name.

DW_DLE_DEBUG_INFO_DUPLICATE
Value is 31. More than one .debug_info section was found.

DW_DLE_DEBUG_INFO_NULL
Value is 32. The .debug_info section is present but an error has occurred while retrieving the content.

DW_DLE_DEBUG_ABBREV_DUPLICATE
Value is 33. More than one .debug_abbrev section was found.

DW_DLE_DEBUG_ABBREV_NULL
Value is 34. The .debug_abbrev section is present but an error has occurred while retrieving the content.

DW_DLE_DEBUG_ARANGES_DUPLICATE
Value is 35. More than one .debug_aranges section was found.

DW_DLE_DEBUG_ARANGES_NULL
Value is 36.The .debug_aranges section is present but an error has occurred while retrieving the content.

DW_DLE_DEBUG_LINE_DUPLICATE
Value is 37. More than one .debug_line section was found.

DW_DLE_DEBUG_LINE_NULL
Value is 38. The .debug_line section is present but an error has occurred while retrieving the content.

DW_DLE_DEBUG_LOC_DUPLICATE
Value is 39. More than one .debug_loc section was found.

DW_DLE_DEBUG_LOC_NULL
Value is 40. The .debug_loc section is present but an error has occurred while retrieving the content.

DW_DLE_DEBUG_MACINFO_DUPLICATE
Value is 41. More than one .debug_macinfo section was found.

DW_DLE_DEBUG_MACINFO_NULL
Value is 42. The .debug_macinfo section is present but an error has occurred while retrieving the content.

DW_DLE_DEBUG_PUBNAMES_DUPLICATE
Value is 43. More than one .debug_pubname section was found.

DW_DLE_DEBUG_PUBNAMES_NULL
Value is 44. The .debug_pubname section is present but an error has occurred while retrieving the content.

DW_DLE_DEBUG_STR_DUPLICATE
Value is 45. More than one .debug_str section was found.

DW_DLE_DEBUG_STR_NULL
Value is 46. The .debug_str section is present but an error has occurred while retrieving the content.

DW_DLE_CU_LENGTH_ERROR
Value is 47. The unit header length of the compilation unit is not valid.

DW_DLE_VERSION_STAMP_ERROR
Value is 48. Incorrect Version Stamp

DW_DLE_ABBREV_OFFSET_ERROR
Value is 49. The .debug_abbrev offset is greater than the size of .debug_abbrev section..

DW_DLE_ADDRESS_SIZE_ERROR
Value is 50. The size of an address on the target machine is not valid.

DW_DLE_DIE_NULL
Value is 52. Dwarf_Die is NULL

DW_DLE_STRING_OFFSET_BAD
The .debug_str offset is greater than the size of .debug_str section.

DW_DLE_DEBUG_LINE_LENGTH_BAD
Value is 54. The length of this .debug_line segment is greater than the size of .debug_line section.

DW_DLE_LINE_PROLOG_LENGTH_BAD
Value is 55. The header length of the .debug_line header is smaller than a recognized form

DW_DLE_LINE_NUM_OPERANDS_BAD
Value is 56. The number of operands given for the line number program opcode is not valid.

DW_DLE_LINE_SET_ADDR_ERROR
Value is 57. The size of the operand specified on DW_LNE_set_address opcode does not match the size of an address on the target machine.

DW_DLE_LINE_EXT_OPCODE_BAD
Value is 58. The line number program extended opcode is not recognized.

DW_DLE_DWARF_LINE_NULL
Value is 59. Dwarf_line is NULL.

DW_DLE_INCL_DIR_NUM_BAD
Value is 60. The directory index of the Dwarf_Line object is out of range.

DW_DLE_LINE_FILE_NUM_BAD
Value is 61. The file index of the Dwarf_Line object is out of range.

DW_DLE_ALLOC_FAIL
Value is 62. The required object could not be allocated.

DW_DLE_NO_CALLBACK_FUNC
Value is 63. The callback function was not specified.

DW_DLE_SECT_ALLOC
Value is 64. Dwarf_Section or Dwarf_P_Section was not allocated.

DW_DLE_FILE_ENTRY_ALLOC
Value is 65. There is an error allocating memory to store file information in the line number table.

"DW_DLE_LINE_ALLOC
Value is 66. Dwarf_Line or Dwarf_P_Line was not allocated.

DW_DLE_FPGM_ALLOC
Value is 67. There is an error allocating memory to store information in the debug_frame section.

DW_DLE_INCDIR_ALLOC
Value is 68. There is an error allocating memory to store directory information in the line number table.

DW_DLE_STRING_ALLOC
Value is 69. String object was not allocated.

DW_DLE_CHUNK_ALLOC
Value is 70. There is an error allocating memory for an internal variable length object.

DW_DLE_CIE_ALLOC
Value is 72. Common Information Entry (CIE) was not allocated.

DW_DLE_FDE_ALLOC
Value is 73. Frame Desciption Entry (FDE) was not allocated.

DW_DLE_REGNO_OVFL
Value is 74. A register number overflow was detected.

DW_DLE_CIE_OFFS_ALLOC
Value is 75. here is an error allocating memory to store CIE_pointer in the .debug_frame section.

DW_DLE_WRONG_ADDRESS
Value is 76. Unable to encode address information in line number table.

DW_DLE_EXTRA_NEIGHBORS
Value is 77. Specifying more than one neighbor is not allowed.

DW_DLE_WRONG_TAG
Value is 78. The input DIE has an unsupported TAG value.

DW_DLE_DIE_ALLOC
Value is 79. Dwarf_Die or Dwarf_P_Die was not allocated.

DW_DLE_PARENT_EXISTS
Value is 80. A parent DIE already exist.

DW_DLE_DBG_NULL
Value is 81. Dwarf_Debug (or Dwarf_P_Debug) object does not exist.

DW_DLE_DEBUGLINE_ERROR
Value is 82. An error has occured while creating .debug_line.

DW_DLE_DEBUGFRAME_ERROR
Value is 83. An error has occured while creating .debug_frame.

DW_DLE_DEBUGINFO_ERROR
Value is 84. An error has occured while creating .debug_info.

DW_DLE_ATTR_ALLOC
Value is 85. Dwarf_Attribute/Dwarf_P_Attribute was not allocated.

DW_DLE_ABBREV_ALLOC
Value is 86. The abbreviation object was not allocated.

DW_DLE_OFFSET_UFLW
Value is 87. Offset is too large to fit in specified container.

DW_DLE_ELF_SECT_ERR
Value is 88. Unknown ELF section found.

DW_DLE_DEBUG_FRAME_LENGTH_BAD
Value is 89. The size of the length field plus the value of length is not an integral multiple of the address size.

DW_DLE_FRAME_VERSION_BAD
Value is 90. The version number of the .debug_frame section is not recognized.

DW_DLE_CIE_RET_ADDR_REG_ERROR
Value is 91. An incorrect register was specified for return address.

DW_DLE_FDE_NULL
Value is 92. Dwarf_Fde/Dwarf_P_Fde object does not exist.

DW_DLE_FDE_DBG_NULL
Value is 93. There is no Dwarf_Debug object associated with the Dwarf_Fde object.

DW_DLE_CIE_NULL
Value is 94. Dwarf_Cie object does not exist.

DW_DLE_CIE_DBG_NULL
Value is 95. There is no Dwarf_Debug associated with the Dwarf_Cie object.

DW_DLE_FRAME_TABLE_COL_BAD
Value is 96. The column in the frame table specified is not valid.

DW_DLE_PC_NOT_IN_FDE_RANGE
Value is 97. PC requested not in address range of FDE.

DW_DLE_CIE_INSTR_EXEC_ERRORspecified
Value is 98. There was an error in executing instructions in CIE.

DW_DLE_FRAME_INSTR_EXEC_ERROR
Value is 99. There was an error in executing instructions in FDE.

DW_DLE_FDE_PTR_NULL
Value is 100. Null Pointer to Dwarf_Fde .

DW_DLE_RET_OP_LIST_NULL
Value is 101. No location to store pointer to Dwarf_Frame_Op

DW_DLE_LINE_CONTEXT_NULL
Value is 102. Dwarf_Line has no context.

DW_DLE_DBG_NO_CU_CONTEXT
Value is 103. dbg has no CU context for dwarf_siblingof().

DW_DLE_DIE_NO_CU_CONTEXT
Value is 104. Dwarf_Die has no CU context.

DW_DLE_FIRST_DIE_NOT_CU
Value is 105. The first DIE in the CU is not a DW_TAG_compilation_unit.

DW_DLE_NEXT_DIE_PTR_NULL
Value is 106. There was an error when moving to next DIE in .debug_info.

DW_DLE_DEBUG_FRAME_DUPLICATE
Value is 107. More than one .debug_frame section was found.

DW_DLE_DEBUG_FRAME_NULL
Value is 108. The .debug_frame section is present but an error has occurred while retrieving the content.

DW_DLE_ABBREV_DECODE_ERROR
Value is 109. There was an error in processing .debug_abbrev section.

DW_DLE_DWARF_ABBREV_NULL
Value is 110. The Dwarf_Abbrev object specified is null.

DW_DLE_ATTR_NULL
Value is 111. The Dwarf_Attribute object specified is null.

DW_DLE_DIE_BAD
Value is 112. There was an error in processing the Dwarf_Die object.

DW_DLE_DIE_ABBREV_BAD
Value is 113. No abbreviation was found for the abbreviation code embedded in the Dwarf_Die object.

DW_DLE_ATTR_FORM_BAD
Value is 114. The attribute form for the attribute is not appropriate.

DW_DLE_ATTR_NO_CU_CONTEXT
Value is 115. There is no CU context for the Dwarf_Attribute object.

DW_DLE_ATTR_FORM_SIZE_BAD
Value is 116. The size of block in attribute value is not valid.

DW_DLE_ATTR_DBG_NULL
Value is 117. There is no Dwarf_Debug object associated with the Dwarf_Attribute object.

DW_DLE_BAD_REF_FORM
Value is 118. The form for the reference attribute is not appropriate.

DW_DLE_ATTR_FORM_OFFSET_BAD
Value is 119. The offset reference attribute is outside current CU.

DW_DLE_LINE_OFFSET_BAD
Value is 120. The offset of lines for the current CU is outside .debug_line.

DW_DLE_DEBUG_STR_OFFSET_BAD
Value is 121. The offset in .debug_str is out of range.

DW_DLE_STRING_PTR_NULL
Value is 122. The pointer to the return string parameter is NULL.

DW_DLE_PUBNAMES_VERSION_ERROR
Value is 123. The version of .debug_pubnames is not recognized.

DW_DLE_PUBNAMES_LENGTH_BAD
Value is 124. The length field in .debug_pubnames section is greater than the size of the section.

DW_DLE_GLOBAL_NULL
Value is 125. The Dwarf_Global specified is null.

DW_DLE_GLOBAL_CONTEXT_NULL
Value is 126. There was no context given for Dwarf_Global.

DW_DLE_DIR_INDEX_BAD
Value is 127. There was an error in the directory index read.

DW_DLE_LOC_EXPR_BAD
Value is 128. The location expression could not be read.

DW_DLE_DIE_LOC_EXPR_BAD
Value is 129. The expected block value for attribute was not found.

DW_DLE_ADDR_ALLOC
Value is 130. There is an error allocating memory for an internal address object.

DW_DLE_OFFSET_BAD
Value is 131. The offset for next compilation-unit in .debug_info is not valid.

DW_DLE_MAKE_CU_CONTEXT_FAIL
Value is 132. The CU context was not created.

DW_DLE_REL_ALLOC
Value is 133. There is an error allocating memory for an internal relocation object.

DW_DLE_ARANGE_OFFSET_BAD
Value is 134. The debug_arange entry has a DIE offset that is larger than the size of the .debug_info section.

DW_DLE_SEGMENT_SIZE_BAD
Value is 135. The segment size should be 0 for MIPS processors.

DW_DLE_ARANGE_LENGTH_BAD
Value is 136. The length field in .debug_aranges section is greater than the size of the section.

DW_DLE_ARANGE_DECODE_ERROR
Value is 137. The aranges do not end at the end of .debug_aranges.

DW_DLE_ARANGES_NULL
Value is 138. The Dwarf_Arange list parameter is NULL.

DW_DLE_ARANGE_NULL
Value is 139. The Dwarf_Arange parameter is NULL.

DW_DLE_NO_FILE_NAME
Value is 140. The file name parameter is NULL.

DW_DLE_NO_COMP_DIR
Value is 141. There was no Compilation directory for compilation-unit.

DW_DLE_CU_ADDRESS_SIZE_BAD
Value is 142. The CU header address size does not match the Elf class.

DW_DLE_INPUT_ATTR_BAD
Value is 143. The attribute on the input DIE is not supported.

DW_DLE_EXPR_NULL
Value is 144. The specified Dwarf_P_Expr object is NULL.

DW_DLE_BAD_EXPR_OPCODE
Value is 145. There is an unsupported DWARF expression opcode specified.

DW_DLE_EXPR_LENGTH_BAD
Value is 146. Unable to create LEB128 constant while constructing DWARF expression.

DW_DLE_BAD_RELOC
Value is 147. Relocation Information found is not correct.

DW_DLE_ELF_GETIDENT_ERROR
Value is 148. There is an error in elf_getident() on object.

DW_DLE_NO_AT_MIPS_FDE
Value is 149. The DIE does not have DW_AT_MIPS_fde attribute.

DW_DLE_NO_CIE_FOR_FDE
Value is 150. There is no CIE specified for FDE.

DW_DLE_DIE_ABBREV_LIST_NULL
Value is 151. There was no abbreviation found for the code in DIE.

DW_DLE_DEBUG_FUNCNAMES_DUPLICATE
Value is 152. More than one .debug_funcnames section was found.

DW_DLE_DEBUG_FUNCNAMES_NULL
Value is 153. The .debug_funcnames section is present but an error has occurred while retrieving the content.

DW_DLE_CANNOT_LOAD_DLLSYM
Value is 154. Unable to load a symbol from DLL to continue processing.

DW_DLE_DEBUG_PUBTYPES_DUPLICATE
Value is 158. More than one .debug_pubtypes section was found.

DW_DLE_DEBUG_PUBTYPES_NULL
Value is 159. The .debug_pubtypes section is present but an error has occurred while retrieving the content.

DW_DLE_DEBUG_VARNAMES_DUPLICATE
Value is 164. More than one .debug_varnames section was found.

DW_DLE_DEBUG_VARNAMES_NULL
Value is 165. The .debug_varnames section is present but an error has occurred while retrieving the content.

DW_DLE_DEBUG_WEAKNAMES_DUPLICATE
Value is 170. More than one .debug_weaknames section was found.

DW_DLE_DEBUG_WEAKNAMES_NULL
Value is 171. The .debug_weaknames section is present but an error has occurred while retrieving the content.

DW_DLE_LOCDESC_COUNT_WRONG
Value is 176. More than one location description found.

DW_DLE_MACINFO_STRING_NULL
Value is 177. The specified macro name is NULL.

DW_DLE_MACINFO_STRING_EMPTY
Value is 178. The specified macro name has zero length.

DW_DLE_MACINFO_INTERNAL_ERROR_SPACE
Value is 179. An error has occurred during construction of .debug_macinfo.

DW_DLE_MACINFO_MALLOC_FAIL
Value is 180. Failed to allocate internal object for writing .debug_macinfo.

DW_DLE_DEBUGMACINFO_ERROR
Value is 181. An error has occured during writing of .debug_macinfo.

DW_DLE_DEBUG_MACRO_LENGTH_BAD
Value is 182. The specified offset is beyond the size of .debug_macinfo.

DW_DLE_DEBUG_MACRO_INTERNAL_ERR
Value is 184. An error has occured during reading of .debug_macinfo.

DW_DLE_DEBUG_MACRO_MALLOC_SPACE
Value is 185. Failed to allocate internal object for reading .debug_macinfo.

DW_DLE_DEBUG_MACRO_INCONSISTENT
Value is 186. Conflicting information found while processing .debug_macinfo.

DW_DLE_DF_NO_CIE_AUGMENTATION
Value is 187. No CIE augmentation was found.

DW_DLE_DF_REG_NUM_TOO_HIGH
Value is 188. The call frame register is too big.

DW_DLE_DF_MAKE_INSTR_NO_INIT
Value is 189. Call frame information has not been initialized.

DW_DLE_DF_NEW_LOC_LESS_OLD_LOC
Value is 190. New instruction offset is less than the old instruction offset.

DW_DLE_DF_POP_EMPTY_STACK
Value is 191. The stack is empty while processing .debug_frame

DW_DLE_DF_ALLOC_FAIL
Value is 192. The internal object for reading .debug_frame was not allocated.

DW_DLE_DF_FRAME_DECODING_ERROR
Value = 193. An error has occurred while reading .debug_frame.

DW_DLE_FLAG_BIT_IDX_BAD
Value = 194. The bit index is out of range.

DW_DLE_RETURN_PTR_NUL
Value = 195. The pointer to the return parameter is NULL.

DW_DLE_LINE_TABLE_ALLOC
Value = 196. Memory allocation failed in creating line number table.

DW_DLE_LINE_TABLE_NULL
Value = 197. The line-number table is empty.

DW_DLE_FILE_ENTRY_BODY
Value = 198. A file entry already exists in the line-number program.

DW_DLE_SECTION_NULL
Value = 200. The given debug section is NULL.

DW_DLE_SECTION_INACTIVE
Value = 201. The given debug section is inactive.

DW_DLE_DEBUG_SRCATTR_ERROR
Value = 202. An error occurred processing .debug_srcattr.

DW_DLE_DEBUG_SRCTEXT_ERROR
Value = 203. An error occurred processing .debug_srctext .

DW_DLE_HASHMAP_ERROR
Value = 204. An internal error occurred while accessing the internal hash table.

DW_DLE_DEBUG_SRCFILES_ERROR
Value = 205. An error occurred processing .debug_srcfiles.

DW_DLE_DEBUG_PPA_ERROR
Value = 206. An error occurred processing .debug_ppa.

DW_DLE_DEBUG_STR_ERROR
Value = 207. An error occurred processing .debug_str.

DW_DLE_DEBUG_XREF_ERROR
Value = 208. An error occurred processing .debug_xref.

DW_DLE_DEBUG_XREF_DUPLICATE
Value = 209. More than one .debug_xref section was found.

DW_DLE_DEBUG_XREF_NULL
Value = 210. The .debug_xref section is present but an error has occurred while retrieving the content.

DW_DLE_SRCATTR_LINE_BAD
Value = 211. The line number found within .debug_srcattr is not greater than 0.

DW_DLE_SRCATTR_OFFSET_BAD
Value = 212. An invalid offset was found in .debug_srcattr.

DW_DLE_SECTION_NAME_NULL
Value = 213. The name of the section is NULL.

DW_DLE_SECTION_NAME_BAD
Value = 214. An unknown debug-section name has been detected.

DW_DLE_LINE_OWNER_BAD
Value = 215. The line-number program does not have a valid owner.

DW_DLE_DEBUG_PPA_DUPLICATE
Value = 216. More than one .debug_ppa section was found.

DW_DLE_DEBUG_PPA_NULL
Value = 217. The .debug_ppa section is present but an error has occurred while retrieving the content.

DW_DLE_DEBUG_SRCFILES_DUPLICATE
Value = 218. More than one .debug_srcfiles section was found.

DW_DLE_DEBUG_SRCFILES_NULL
Value = 219. The .debug_srcfiles section is present but an error has occurred while retrieving the content.

DW_DLE_DEBUG_SRCINFO_DUPLICATE
Value = 220. More than one .debug_srcinfo section was found.

DW_DLE_DEBUG_SRCINFO_NULL
Value = 221. The .debug_srctext section is present but an error has occurred while retrieving the content.

DW_DLE_DEBUG_SRCTEXT_DUPLICATE
Value = 222. More than one .debug_srctext section was found.

DW_DLE_DEBUG_SRCTEXT_NULL
Value = 223. The .debug_srctext section is present but an error has occurred while retrieving the content.

DW_DLE_DEBUG_SRCATTR_DUPLICATE
Value = 224. More than one .debug_srcattr section was found.

DW_DLE_DEBUG_SRCATTR_NULL
Value = 225. The .debug_srcattr section is present but an error has occurred while retrieving the content.

DW_DLE_DEBUG_SRCATTR_DECODE
Value = 226. An internal error occurred while processing .debug_srcattr section.

DW_DLE_SRCFRAG_NULL
Value = 227. The source fragment object is NULL.

DW_DLE_ELF_STRING_NULL
Value = 228. A NULL string cannot be added into an ELF section.

DW_DLE_ELF_STRING_ALLOC
Value = 229. The memory allocation failed while creating a string in an ELF section.

DW_DLE_ELF_SYMBOL_NULL
Value = 230. The ELF-symbol name is NULL.

DW_DLE_ELF_SYMBOL_BAD
Value = 231. The ELF-symbol name is invalid.

DW_DLE_ELF_SYMBOL_ALLOC
Value = 232. The memory allocation failed when creating an ELF symbol.

DW_DLE_LINE_INFO_NULL
Value = 233. The line-number program contains no information.

DW_DLE_DEBUG_RANGES_DUPLICATE
Value = 234. More than one .debug_ranges section was found.

DW_DLE_DEBUG_RANGES_NULL
Value = 235. The .debug_ranges section is present but an error has occurred while retrieving the content.

DW_DLE_DEBUG_INFO_RELOC_DUPLICATE
Value = 236. More than one relocation section for .debug_info was found.

DW_DLE_DEBUG_INFO_RELOC_NULL
Value = 237. The .rel.debug_info section is present but an error has occurred while retrieving the content.

DW_DLE_DEBUG_LINE_RELOC_DUPLICATE
Value = 238. More than one relocation section for .debug_line was found.

DW_DLE_DEBUG_LINE_RELOC_NULL
Value = 239. The .rel.debug_line section is present but an error has occurred while retrieving the content.

DW_DLE_LINE_CONTEXT_STACK_FULL
Value = 240. The gap stack becomes full while building line context.

DW_DLE_ELF_WRITE_ERROR
Value = 241. An error occurred when writing to ELF.

DW_DLE_NAME_NULL
Value = 242. The given name is NULL.

DW_DLE_NAME_EMPTY
Value = 243. The given name is empty.

DW_DLE_ELF_NULL
Value = 244. The ELF descriptor is NULL.

DW_DLE_ELF_MACHINE_UNKNOWN
Value = 245. The hardware architecture is unknown.

DW_DLE_PC_LOCN_NULL
Value = 246. The Dwarf_PC_Locn object is NULL.

DW_DLE_SUBPGM_LOCN_NULL
Value = 247. The Dwarf_Subpgm_Locn object is NULL.

DW_DLE_FILE_INDEX_BAD
Value = 248. The file index within the line-number program is out of range.

DW_DLE_GET_LINE_FAILED
Value = 249. An error occurred during the retrieval of one or more source lines.

DW_DLE_CANNOT_LOAD_DLL
Value = 250. Unable to load the required DLL to continue processing.

DW_DLE_RANGES_DECODE_ERROR
Value = 251. The range-list entry extends beyond the end of .debug_ranges.

DW_DLE_CODESET_INVALID
Value = 252. The given codeset ID is not valid.

DW_DLE_CODESET_CONVERSION_ERROR
Value = 253. There was an error converting between codesets.

DW_DLE_STRING_NULL
Value = 254. The Dwarf_String object is NULL.

DW_DLE_PROGRAM_OBJECT_EDIT_NO
Value = 255. Program object must be bounded with EDIT=YES.

DW_DLE_CANNOT_FIND_FULLPATH
Value = 256. Unable to resolve full path name for the given file name.

DW_DLE_PROGRAM_OBJECT_PROCESS_ERROR
Value = 257. An internal error has occurred while processing the program object.

DW_DLE_LOC_LIST_DECODE_ERROR
Value = 258. Location list entry has reached the end of .debug_loc section, but has incomplete data.

DW_DLE_ASM_DIR_GEN_ERROR
Value = 259. Unable to generate an assembly directive representing Dwarf debug data.

DW_DLE_DWARF_SPEC_VIOLATION
Value = 260. Debug entity is not supported by the given Dwarf Spec version or the given extension mode.

dwarf_error_reset operation

The `dwarf_error_reset` operation resets the error code within a valid `Dwarf_Error` object to `DW_DLE_NE` (no error).

If the error parameter is NULL or does not contain a valid `Dwarf_Error` object, this operation will do nothing.

Prototype

```
void dwarf_error_reset (
  Dwarf_Error*           error);
```

Parameters

error
Input/output. This accepts or returns a `Dwarf_Error` object.

Initialization and termination consumer operations

This section contains a list of APIs related to creating and terminating `libdwarf` consumer objects.

dwarf_elf_init_b operation

Given an elf descriptor obtained from ELF operations, this operation creates and initializes a `libdwarf` consumer instance. This operation replaces the functionality of the `dwarf_elf_init` operation, and provides the added ability to combine multiple `libdwarf` consumer instances into a single one.

If the given or returned object already exists, then `dwarf_elf_init_b` creates a new object by merging the existing content with the new content. That is, if `ret_dbg` contains non-NULL `libdwarf` object, then this operation will create a new `libdwarf` object derived from `elfptr` and merge it into the existing `libdwarf` object.

If the given or returned DWARF object is NULL, then a completely new object is created. In this case, `dwarf_elf_init_b` behaves the same as the core `libdwarf` operation `dwarf_elf_init`.

Prototype

```
int dwarf_elf_init_b(
  Elf*                elfptr,
  Dwarf_Unsigned      access,
  Dwarf_Handler       errhand,
  Dwarf_Ptr           errarg,
  Dwarf_Debug*        ret_dbg,
  Dwarf_Error*        error);
```

Parameters

elfptr
Input. This accepts the elf descriptor from ELF operations. When the `dwarf_elf_init_b` operation is invoked, it assumes control of this descriptor, which prevents the user from using or referencing this elf descriptor.

access
Input. This accepts the file access method:

- For DWARF consumer operations, it is DW_DLC_READ read only access.

- For DWARF producer operations, it is DW_DLC_WRITE write only access.

errhand
Input. This accepts the default error handler if it is used. If default error handler is not used, it accepts the NULL value.

errarg
Input. When an error condition is triggered within any of the DWARF consumer operations, the errhand parameter accepts this object.

ret_dbg
Input/output. If *ret_dbg is NULL, then this routine is identical to dwarf_elf_init. If *ret_dbg is a valid libdwarf instance, this dwarf debug information will be merged with the dwarf debug information embedded within elfptr. The operation then initializes a new libdwarf instance containing the merged dwarf debug information. The user should deallocate this after use.

error
Input/output. This accepts or returns a Dwarf_Error object.

Return values

DW_DLV_OK
A valid libdwarf consumer instance is returned.

DW_DLV_NO_ENTRY
DWARF debug sections are not present in the given Elf object.

DW_DLV_ERROR

DW_DLE_ELF_NULL
Given Elf object is NULL

DW_DLE_RETURN_PTR_NULL
Given 'ret_dbg' is NULL

DW_DLE_INIT_ACCESS_WRONG
Incorrect file access method. See dwarfInitFlags

DW_DLE_DBG_ALLOC
Unable to allocate memory for creating libdwarf consumer instance

DW_DLE_ELF_GETIDENT_ERROR
Unable to retrieve ELF Identification

DW_DLE_ELF_GETEHDR_ERROR
Unable to retrieve ELF header.

DW_DLE_ALLOC_FAIL
Unable to allocate memory for creating internal objects

DW_DLE_ELF_GETSHDR_ERROR
Unable to retrieve ELF section header

DW_DLE_ELF_STRPTR_ERROR
Unable to retrieve name of ELF section

DW_DLE_DEBUG_INFO_DUPLICATE
More than one .debug_info section was found.

DW_DLE_DEBUG_INFO_NULL
Either the .debug_info section does not exist or it is empty.

DW_DLE_DEBUG_ABBREV_DUPLICATE
More than one .debug_abbev section was found.

DW_DLE_DEBUG_ABBREV_NULL
Either the .debug_abbrev section does not exist or it is empty.

DW_DLE_DEBUG_ARANGES_DUPLICATE
More than one .debug_aranges section was found.

DW_DLE_DEBUG_ARANGES_NULL
The .debug_aranges section exists but it is empty.

DW_DLE_DEBUG_RANGES_DUPLICATE
More than one .debug_ranges section was found.

DW_DLE_DEBUG_RANGES_NULL
The .debug_ranges section exists but it is empty.

DW_DLE_DEBUG_LINE_DUPLICATE
More than one .debug_line section was found.

DW_DLE_DEBUG_LINE_NULL
The .debug_line section exists but it is empty.

DW_DLE_DEBUG_FRAME_DUPLICATE
More than one .debug_frame or .eh_frame section was found.

DW_DLE_DEBUG_FRAME_NULL
The .debug_frame section exists but it is empty.

DW_DLE_DEBUG_LOC_DUPLICATE
More than one .debug_loc section was found.

DW_DLE_DEBUG_LOC_NULL
The .debug_loc section exists but it is empty.

DW_DLE_DEBUG_PUBNAMES_DUPLICATE
More than one .debug_pubnames section was found.

DW_DLE_DEBUG_PUBNAMES_NULL
The .debug_pubnames section exists but it is empty.

DW_DLE_DEBUG_PUBTYPES_DUPLICATE
More than one .debug_pubtypes section was found.

DW_DLE_DEBUG_PUBTYPES_NULL
The .debug_pubtypes section exists but it is empty.

DW_DLE_DEBUG_STR_DUPLICATE
More than one .debug_str section was found.

DW_DLE_DEBUG_STR_NULL
The .debug_str section exists but it is empty.

DW_DLE_DEBUG_FUNCNAMES_DUPLICATE
More than one .debug_funcnames section was found.

DW_DLE_DEBUG_FUNCNAMES_NULL
The .debug_funcnames section exists but it is empty.

DW_DLE_DEBUG_VARNAMES_DUPLICATE
More than one .debug_varnames section was found.

DW_DLE_DEBUG_VARNAMES_NULL
The .debug_varnames section exists but it is empty.

DW_DLE_DEBUG_WEAKNAMES_DUPLICATE
More than one .debug_weaknames section was found.

DW_DLE_DEBUG_WEAKNAMES_NULL
The .debug_weaknames section exists but it is empty.

DW_DLE_DEBUG_MACINFO_DUPLICATE
More than one .debug_macinfo section was found.

DW_DLE_DEBUG_MACINFO_NULL
The .debug_macinfo section exists but it is empty.

DW_DLE_DEBUG_PPA_DUPLICATE
More than one .debug_ppa section was found.

DW_DLE_DEBUG_PPA_NULL
The .debug_ppa section exists but it is empty.

DW_DLE_DEBUG_SRCFILES_DUPLICATE
More than one `.debug_srcfiles` section was found.

DW_DLE_DEBUG_SRCFILES_NULL
The `.debug_srcfiles` section exists but it is empty.

Cleanups

Do not call `elf_end` until after `dwarf_finish` is called. `ret_dbg` can be deallocated by calling `dwarf_finish`, as shown in the following code block:

```
Elf*        elf;
Dwarf_Debug dbg;
dwarf_elf_init_b (elf, ..., &dbg, ...);
...
// 'elf' must be saved before 'dbg' is terminated
dwarf_get_elf (dbg, &elf, ...);

// terminate 'dbg'
dwarf_finish (dbg, error);

// terminate 'elf' (optional)
elf_end(elf);
```

Note: To simplify the example, only the relevant parameters are found in the above code. Unlisted parameters are represented by ellipses(...).

dwarf_goff_init_with_csvquery_token operation

Given a CSVQUERY token, this API creates and initializes a libdwarf consumer instance.

Prototype

```
int
  dwarf_goff_init_with_csvquery_token(
    void*               eptoken,
    Dwarf_Handler       errhand,
    Dwarf_Ptr           errarg,
    Dwarf_Addr          ccode_addr,
    Dwarf_Debug*        ret_dbg,
    Dwarf_Error*        error);
```

Parameters

eptoken
Input. The CSVQUERY token.

errhand
Input. NULL if the default error handler is used.

errarg
Input. When an error condition is triggered within any of the DWARF consumer operations, this object is passed to the `errhand` parameter.

code_addr
Input. A real C_CODE address used for relocating all address related to C_CODE

ret_dbg
Output. `libdwarf` consumer instance.

error
Error. This accepts or returns a `Dwarf_Error` object.

Return values

DW_DLV_OK
A valid libdwarf consumer instance is returned.

DW_DLV_NO_ENTRY
Unable to initialize binder API

DWARF debug sections are not present in the given GOFF object.

DW_DLV_ERROR

DW_DLE_RETURN_PTR_NULL
Given `ret_dbg` is NULL

DW_DLE_DBG_ALLOC
Unable to allocate memory for *ret_dbg

DW_DLE_ALLOC_FAIL
Unable to allocate memory for internal objects

DW_DLE_PROGRAM_OBECT_EDIT_NO
The program object is bound with EDIT=NO.

DW_DLE_PROGRAM_OBJECT_PROCESS_ERROR
Unable to process the input program object.

Cleanups

```
dwarf_goff_init_with_csvquery (eptoken, ..., &dbg, &err);
...
// terminate 'dbg'
dwarf_finish (dbg, error);
```

Note: To simplify the example, only the relevant parameters are found in the above code. Unlisted parameters are represented by ellipses(...).

dwarf_goff_init_with_PO_filename operation

Given a GOFF program object filename, this API creates and initializes a libdwarf consumer instance.

Prototype

```
int
  dwarf_goff_init_with_PO_filename(
    char*               filename,
    Dwarf_Handler       errhand,
    Dwarf_Ptr           errarg,
    Dwarf_Addr          ccode_addr,
    Dwarf_Debug*        ret_dbg,
    Dwarf_Error*        error);
```

Parameters

filename
Input. GOFF program object file name. It must be encoded in IBM-1047.

errhand
Input. NULL if the default error handler is used.

errarg
Input. When an error condition is triggered within any of the DWARF consumer operations, the `errhand` parameter accepts this object.

code_addr
Input. A real C_CODE address used for relocating all address related to C_CODE

ret_dbg
Output. `libdwarf` consumer instance.

error
Error. This accepts or returns a `Dwarf_Error` object.

Return values

DW_DLV_OK
A valid libdwarf consumer instance is returned.

DW_DLV_NO_ENTRY
Unable to initialize binder API

DWARF debug sections are not present in the given GOFF object.

DW_DLV_ERROR

DW_DLE_RETURN_PTR_NULL
Given 'ret_dbg' is NULL

DW_DLE_DBG_ALLOC
Unable to allocate memory for *ret_dbg

DW_DLE_ALLOC_FAIL
Unable to allocate memory for internal objects

DW_DLE_FNO
Unable to open `filename`.

DW_DLE_NOB
`filename` is 0 length or it is not a valid GOFF program object.

DW_DLE_FNR
Input `filename` contains invalid characters.

DW_DLE_CANNOT_FIND_FULLPATH
Unable to determine absolute path for `filename`. Make sure all paths leading to `filename` have read and execute permission set.

DW_DLE_PROGRAM_OBECT_EDIT_NO
The program object is bound with EDIT=NO.

DW_DLE_PROGRAM_OBJECT_PROCESS_ERROR
Unable to process the input program object.

Cleanups

```
Dwarf_Debug dbg;
dwarf_goff_init_with_PO_filename ("a.out", ..., &dbg, &err);
...
// terminate 'dbg'
dwarf_finish (dbg, error);
```

Note: To simplify the example, only the relevant parameters are found in the above code. Unlisted parameters are represented by ellipses(...).

dwarf_raw_binary_init operation

The `dwarf_raw_binary_init` operation initializes libdwarf consumer instance with all DWARF section provided in raw binary format.

Prototype

```
int dwarf_raw_binary_init(
    Dwarf_Block*        dwf_data,
    Dwarf_Bool          bigendian,
    Dwarf_Bool          is_64bit,
    Dwarf_Handler       errhand,
    Dwarf_Ptr           errarg,
    Dwarf_Debug*        ret_dbg,
    Dwarf_Error*        error);
```

Parameters

dwf_data
Input. This is an array of DW_SECTION_NUM_SECTIONS elements. Each element is of type Dwarf_Block. bl_data points to the start of the debug section, and bl_len is the size of the debug section.

bigendian
Input. True if the debug sections are encoded in big endian.

is_64bit
Input. True if the debug sections are 64-bit DWARF.

errhand
Input. Error handler. NULL if the default error handler is used.

errarg
Input. When an error condition is triggered within any of the DWARF Consumer APIs, this object is passed into errhand specified above.

ret_dbg
Output. This is the libdwarf consumer instance.

error
Input/Output. Error. This accepts and returns the Dwarf_Error object.

Return values

DW_DLV_OK
A valid libdwarf consumer instance is returned.

DW_DLV_NO_ENTRY

- Unable to initialize the binder API.
- DWARF debug sections are not present in the given GOFF object.

DW_DLV_ERROR

DW_DLE_RETURN_PTR_NULL
The given ret_dbg is NULL.

DW_DLE_DBG_ALLOC
Cannot allocate memory for ret_dbg.

DW_DLE_ALLOC_FAIL
Cannot allocate memory for internal objects.

dwarf_set_codeset operation

The dwarf_set_codeset operation specifies the codeset for all the strings (character arrays) that will be passed to the libdwarf consumer operations. This operation overrides the default codeset ISO8859-1. This operation is not available in the IBM CICS® environment.

Prototype

```
int dwarf_set_codeset(
  Dwarf_Debug      dbg,
  const __ccsid_t  codeset_id,
  __ccsid_t*       prev_cs_id,
  Dwarf_Error*     error);
```

Parameters

dbg
Input. This libdwarf consumer instance accepts the Dwarf_Debug object.

codeset_id
Input. The CCSID of the strings that will be processed by the libdwarf consumer operations.

prev_cs_id
Output. The previous CCSID specified.

error
Input/Output. Error. This accepts and returns the Dwarf_Error object.

Return values

DW_DLV_OK
The specified codeset ID is valid. All future calls to libdwarf consumer operations will use this encoding for the input/output strings.

DW_DLV_NO_ENTRY
Never.

DW_DLV_ERROR

DW_DLE_DBG_NULL
The given Dwarf_Debug object is NULL

DW_DLE_CODESET_INVALID
Either the given CCSID is invalid or the operation is being used in CICS environment

DW_DLE_CODESET_CONVERSION_ERROR
The operation is unable to find a suitable conversion table to support conversion of the default CODESET (ISO8859-1) to the specified codeset.

dwarf_super_elf_init operation

Given a debug side file that is generated by COBOL compilers V6.2 and later releases, the dwarf_super_elf_init operation creates and initializes a libdwarf consumer instance.

If the given or returned object exists, dwarf_super_elf_init creates a new object by merging the new content with the existing content. That is, if ret_dbg contains non-NULL libdwarf object, this operation creates a new libdwarf object that is derived from the filename parameter and merges it into the existing libdwarf object.

If the given or returned DWARF object is NULL, a new object is created.

Prototype

```
int dwarf_super_elf_init (
    char*               filename,
    Dwarf_Unsigned      access,
    Dwarf_Handler       errhand,
    Dwarf_Ptr           errarg,
    Dwarf_Unsigned      cu_num,
    char*               MD5List,
    Dwarf_Addr*         ccode_addr_lst,
    Dwarf_Debug*        ret_dbg,
    Dwarf_Error*        error);
```

Parameters

filename
Input. This accepts the debug side file name that is generated by COBOL compilers V6.2 and later releases (Super ELF).

access
Input. This accepts the file access method.

errhand
Input. This accepts the default error handler if it is used. If default error handler is not used, it accepts the NULL value.

errarg
Input. When an error condition is triggered within any of the DWARF consumer operations, the errhand parameter accepts this object.

cu_num
Input. Number of CU(s) for which user want to extract Dwarf.

MD5List
Input. An array of MD5 signatures, one for each CU for which user want to extract Dwarf.

ccode_addr_lst
Input. An array of loaded C_CODE addresses, one for each CU for which user want to extract Dwarf.

ret_dbg

Input/output. If *ret_dbg is NULL, this routine initializes a libdwarf consumer instance. If *ret_dbg is a valid libdwarf instance, this dwarf debug information is merged with the dwarf debug information that is derived from the filename parameter.

error
Input/output. Error. This accepts and returns the Dwarf_Error object.

Return values

DW_DLV_OK
A valid libdwarf consumer instance is returned.

DW_DLV_NO_ENTRY
DWARF debug sections are not present in the given debug side file.

DW_DLV_ERROR

Cleanups

Do not call elf_end until after dwarf_finish is called. ret_dbg can be deallocated by calling dwarf_finish, as shown in the following code block:

```
Elf* elf;
Dwarf_Debug dbg;
dwarf_super_elf_init (..., &dbg, ...);
...
// 'elf' must be saved before 'dbg' is terminated
dwarf_get_elf (dbg, &elf, ...);

// terminate 'dbg'
dwarf_finish (dbg, error);

// terminate 'elf' (optional)
elf_end(elf);
```

Note: To simplify the example, only the relevant parameters are found in the previous code. Unlisted parameters are represented by ellipses(...).

ELF symbol table and section consumer operations

This section contains a list of APIs related to accessing information from the ELF symbol table (.symtab section). These APIs are only applicable to libdwarf consumer objects that are initialized with libelf objects.

ELF symbol table

Example of a typical ELF symbol table.

In this example, the .text section (Sym 1) contains information for relocating the addresses within the ELF object file. All relocatable addresses within the ELF object file have offsets relative to the top of the .text section. The value field corresponds to the PPA2 address of this compilation unit. In this example the PPA2 block is 0x1d8 bytes from the top of the .text section. The last 32 byte of the name

field contains a string version of the 16-byte MD5 signature that is found in the object file. For the location of the MD5 signature in the object file, refer to the *z/OS Language Environment Vendor Interfaces*.

```
Sect 18 .symtab symtab off=0x57e 0x6ae size=304 addr=0x0 align=1
        flag=0x0 [---] esize=16 info=19 link=17
  String table = ".strtab"
  Sym 0: value= 0x000, size= 0 sect= undef, type= none, bind= local, name=
  Sym 1: value= 0x1d8, size= 0 sect= .text, type= none, bind= local,
         name= .ppa2_b_546754C452AA8DEB123556EDD3656CC4
  Sym 2: value= 0x000, size= 0 sect= abs, type= file, bind= local, name= a.c
  Sym 3: value= 0x000, size= 1 sect= .debug_info, type= sect, bind= local, name=
  Sym 4: value= 0x000, size= 1 sect= .debug_line, type= sect, bind= local, name=
```

Note: Refer to *ELF Application Binary Interface Supplement* for the layout of the symbol-table entry.

dwarf_elf_symbol_index_list operation

The `dwarf_elf_symbol_index_list` operation retrieves an index entry from the ELF symbol table for a given symbol name.

Prototype

```
int dwarf_elf_symbol_index_list(
  Dwarf_Debug          dbg,
  char *               sym_name,
  Dwarf_Unsigned**     ret_elf_symilst,
  Dwarf_Unsigned*      ret_elf_symcnt,
  Dwarf_Error*         error);
```

Parameters

dbg
Input. This accepts a `libdwarf` consumer object.

sym_name
Input. This accepts the name of an ELF symbol.

ret_elf_symilst
Output. This returns a list of ELF-symbol indexes that match the given name.

ret_elf_symcnt
Output. This returns the count of the ELF-symbol indexes in the list.

error
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

The `dwarf_elf_symbol_index_list` operation returns `DW_DLV_NO_ENTRY` if the `sym_name` value is not found in the ELF symbol table.

Memory allocation

You can deallocate the parameters as required.

Example: The following example is a code fragment that deallocates the `ret_elf_symilst` parameter:

```
if (dwarf_elf_symbol_index_list (dbg,...&ret_elf_symilst, &ret_elf_symcnt, &err)
  == DW_DLV_OK) {
  for (i=0; i<*ret_elf_symcnt; i++)
    dwarf_dealloc (ret_elf_symilst[i], DW_DLA_ADDR);
  dwarf_dealloc (ret_elf_symilst, DW_DLA_LIST);
}
```

Note: To simplify the example, only the relevant parameters are found in the above code. Unlisted parameters are represented by ellipses(...).

dwarf_elf_symbol operation

The dwarf_elf_symbol operation retrieves ELF symbol table-entry data for a given index.

Prototype

```
int dwarf_elf_symbol(
  Dwarf_Debug          dbg,
  Dwarf_Unsigned       elf_symidx,
  char **              ret_sym_name,
  Dwarf_Addr*          ret_sym_value,
  Dwarf_Unsigned*      ret_sym_size,
  unsigned char*       ret_sym_type,
  unsigned char*       ret_sym_bind,
  unsigned char*       ret_sym_other,
  Dwarf_Signed*        ret_sym_shndx,
  Dwarf_Error*         error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer object.

elf_symidx
Input. This accepts the ELF index.

ret_sym_name
Output. This returns the name of the ELF symbol.

ret_sym_value
Output. This returns the value of the ELF symbol.

ret_sym_size
Output. This returns the size of the ELF symbol.

ret_sym_type
Output. This returns the type of the ELF symbol.

ret_sym_bind
Output. This returns the bind of the ELF symbol.

ret_sym_other
Output. This returns any other required value of the ELF symbol.

ret_sym_shndx
Output. This returns the shndx of the ELF symbol.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_elf_symbol operation returns DW_DLV_NO_ENTRY if:

- The ELF symbol table does not exist
- The value of elf_symidx is out of range

dwarf_elf_section operation

The dwarf_elf_section operation retrieves the ELF section for a given index.

Prototype

```
int dwarf_elf_section(
  Dwarf_Debug          dbg,
  Dwarf_Signed         elf_shndx,
```

```
  Elf_Scn**            ret_elf_scn,
  Dwarf_Error*         error);
```

Parameters

dbg
Input. This accepts a `libdwarf` consumer object.

elf_shndx
Input. This accepts the ELF-index section.

ret_elf_scn
Output. This returns the ELF-section object.

error
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

The `dwarf_elf_section` operation returns `DW_DLV_NO_ENTRY` if `elf_shndx` is out of range.

Generalized DIE-section consumer APIs

In standard DWARF, there is only one type of DIE-section, namely `.debug_info` section. IBM provides extensions to DWARF by introducing additional DIE-sections (for example, `debug_srcfiles`). This chapter contains a list of APIs related to navigating between these DIE-sections.

IBM Extensions to DWARF DIE-sections

This section provides a list of DIE-sections introduced by IBM.

The extended sections are:

- .debug_ppa
- .debug_srcfiles
- .debug_xref

Dwarf_section_type enumeration

This enumeration contains a list of supported DWARF sections supported by CDA. These values can be used within the APIs for specifying a particular DWARF section.

Type definition

```
typedef enum Dwarf_section_type_s {
  DW_SECTION_DEBUG_INFO        =  0,
  DW_SECTION_DEBUG_LINE        =  1,
  DW_SECTION_DEBUG_ABBREV      =  2,
  DW_SECTION_DEBUG_FRAME       =  3,
  DW_SECTION_EH_FRAME          =  4,
  DW_SECTION_DEBUG_ARANGES     =  5,
  DW_SECTION_DEBUG_RANGES      =  6,
  DW_SECTION_DEBUG_PUBNAMES    =  7,
  DW_SECTION_DEBUG_PUBTYPES    =  8,
  DW_SECTION_DEBUG_STR         =  9,
  DW_SECTION_DEBUG_FUNCNAMES   =  10,
  DW_SECTION_DEBUG_VARNAMES    =  11,
  DW_SECTION_DEBUG_WEAKNAMES   =  12,
  DW_SECTION_DEBUG_MACINFO     =  13,
  DW_SECTION_DEBUG_LOC         =  14,
  DW_SECTION_DEBUG_PPA         =  15,
  DW_SECTION_DEBUG_SRCFILES    =  16,
  DW_SECTION_DEBUG_SRCTEXT     =  17,
  DW_SECTION_DEBUG_SRCATTR     =  18,
  DW_SECTION_DEBUG_XREF        =  19,
  DW_SECTION_NUM_SECTIONS
```

```
} Dwarf_section_type;
```

Only the following DWARF sections are DIE-sections, and can be used for DIE-section APIs:

```
DW_SECTION_DEBUG_INFO
DW_SECTION_DEBUG_PPA
DW_SECTION_DEBUG_SRCFILES
DW_SECTION_DEBUG_XREF
```

Members

```
Dwarf_section_type           ELF section name   GOFF class name
============================================================
DW_SECTION_DEBUG_INFO        .debug_info        D_INFO
DW_SECTION_DEBUG_LINE        .debug_line        D_LINE
DW_SECTION_DEBUG_ABBREV      .debug_abbrev      D_ABREV
DW_SECTION_DEBUG_FRAME       .debug_frame       D_FRAME
DW_SECTION_EH_FRAME          .eh_frame          n/a
DW_SECTION_DEBUG_ARANGES     .debug_aranges     D_ARNGE
DW_SECTION_DEBUG_RANGES      .debug_ranges      D_RNGES
DW_SECTION_DEBUG_PUBNAMES    .debug_pubnames    D_PBNMS
DW_SECTION_DEBUG_PUBTYPES    .debug_pubtypes    D_PTYPES
DW_SECTION_DEBUG_STR         .debug_str         D_STR
DW_SECTION_DEBUG_FUNCNAMES   .debug_funcnames   D_SFUNC
DW_SECTION_DEBUG_VARNAMES    .debug_varnames    D_SVAR
DW_SECTION_DEBUG_WEAKNAMES   .debug_weaknames   D_WEAK
DW_SECTION_DEBUG_MACINFO     .debug_macinfo     D_MACIN
DW_SECTION_DEBUG_LOC         .debug_loc         D_LOC
DW_SECTION_DEBUG_PPA         .debug_ppa         D_PPA
DW_SECTION_DEBUG_SRCFILES    .debug_srcfiles    D_SRCF
DW_SECTION_DEBUG_SRCTEXT     .debug_srctext     D_SRCTXT
DW_SECTION_DEBUG_SRCATTR     .debug_srcattr     D_SRCATR
DW_SECTION_DEBUG_XREF        .debug_xref        D_XREF
DW_SECTION_DEBUG_TYPE        .debug_types       D_TYPES
```

Dwarf_section_content enumeration

This section provides a list of DWARF section content types supported by CDA.

Type definition

```
typedef enum {
   DW_SECTION_IS_DEBUG_DATA     =  0,
   DW_SECTION_IS_REL            =  1,
   DW_SECTION_IS_RELA           =  2
} Dwarf_section_content;
```

Members

The following members are supported:

DW_SECTION_IS_DEBUG_DATA
Use this to retrieve DWARF section that carry debug information. This is applicable to both ELF object and GOFF program object.

DW_SECTION_IS_REL
Use this to retrieve ELF relocation section for the corresponding DWARF section specified by the `Dwarf_section_type` enumeration. This is applicable to ELF object only.

DW_SECTION_IS_RELA
Use this to retrieve ELF relocation section with addend for the corresponding DWARF section specified by the `Dwarf_section_type` enumeration. This is applicable to ELF object only.

dwarf_debug_section operation

The dwarf_debug_section operation accesses a debug section by specifying the Dwarf_section_type and the Dwarf_section_content enumerations.

The operation supports both debug data, and debug data relocation sections.

Prototype

```
int dwarf_debug_section(
  Dwarf_Debug             dbg,
  Dwarf_section_type      type,
  Dwarf_section_content   content,
  Dwarf_Section*          ret_section,
  Dwarf_Error*            error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer enumeration.

type
Input. This accepts the debug-section type.

content
Input. This accepts the debug-section content.

ret_section
Output. This returns the Dwarf_Section enumeration.

error
Input/output. This accepts or returns the Dwarf_Error enumeration.

Return values

dwarf_debug_section returns DW_DLV_NO_ENTRY if the debug section does not exist.

dwarf_debug_section_name operation

The dwarf_debug_section_name operation queries the name of a given debug section.

The operation supports both debug data, and debug data relocation sections.

Prototype

```
int dwarf_debug_section_name(
  Dwarf_Debug          dbg,
  Dwarf_Section        section,
  char **              ret_name,
  Dwarf_Error*         error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer object.

section
Input. This accepts the Dwarf_Section object.

ret_name
Output. This returns the debug-section name.

error
Input/output. This accepts or returns the Dwarf_Error object.

Example: Parameter deallocation

You can deallocate the parameters as required.

The following example is a code fragment that deallocates the `ret_name` parameter:

```
if (dwarf_debug_section_name (dbg,...&ret_name, &err)
  == DW_DLV_OK) {
  dwarf_dealloc (ret_name, DW_DLA_STRING);
}
```

Note: To simplify the example, only the relevant parameters are found in the above code. Unlisted parameters are represented by ellipses (...).

For more information about deallocating the `error` parameter, see *Consumer Library Interface to DWARF*, by the UNIX International Programming Languages Special Interest Group.

dwarf_next_unit_header operation

The `dwarf_next_unit_header` operation functions like the `dwarf_next_cu_header` operation; in addition it queries information in the unit header of any DIE-format section.

The next invocation of this operation will query the information in the first unit header.

Note: For more information about the `dwarf_next_cu_header` operation, see section 5.2.2 in *A Consumer Library Interface to DWARF*.

Subsequent invocations of this operation pass through the `.debug_info` section. When at the end of the section, the next invocation will return to the start of the section and will query the information in the first unit header.

The related operation is `dwarf_reset_unit_header`. This operation resets the entry point of the `dwarf_next_header` to the beginning of the section.

Prototype

```
int dwarf_next_unit_header(
  Dwarf_Debug        dbg,
  Dwarf_Section      section,
  Dwarf_Unsigned*    ret_unit_length,
  Dwarf_Half*        ret_version,
  Dwarf_Off*         ret_abbrev_ofs,
  Dwarf_Half*        ret_addr_size,
  Dwarf_Off*         ret_next_hdr_ofs
  Dwarf_Error*       error);
```

Parameters

dbg
Input. This accepts a `libdwarf` consumer object.

section
Input. This accepts a `Dwarf_Section` object.

ret_unit_length
Output. This returns the unit length.

ret_version
Output. This returns the DWARF version.

ret_abbrev_ofs
Output. This returns the offset of related `.debug_abbr` information.

ret_addr_size
Output. This returns the address size.

ret_next_hdr_ofs
Output. This returns the offset to the next unit header in the section.

error
Input/output. This accepts or returns the Dwarf_Error object.

Note: All return parameters can be NULL except ret_next_hdr_ofs.

Return values

dwarf_next_unit_header returns DW_DLV_NO_ENTRY if there are no more unit headers in the .debug_info section.

dwarf_reset_unit_header operation

The dwarf_reset_unit_header operation directs subsequent calls to the dwarf_next_unit_header operation to search for the first header unit within the debug section specified.

A subsequent call to dwarf_next_unit_header retrieves information from the first unit header within the specified section.

If the section parameter refers to the .debug_info section, a subsequent call to dwarf_next_cu_header retrieves information from the first unit header within that section.

Prototype

```
int dwarf_reset_unit_header (
  Dwarf_Debug        dbg,
  Dwarf_Section      section,
  Dwarf_Error*       error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer object.

section
Input. This accepts a Dwarf_Section object.

error
Input/output. This accepts or returns the Dwarf_Error object.

DIE locating consumer operations

This section contains a list of APIs for locating a specific DIE within a given DIE section.

dwarf_rootof operation

The dwarf_rootof operation locates the root DIE of a given DIE-format section unit the section unit's header offset.

Prototype

```
int dwarf_rootof(
  Dwarf_Section       section,
  Dwarf_Off           unit_hdr_offset,
  Dwarf_Die*          ret_rootdie,
  Dwarf_Error*        error);
```

Parameters

section
Input. This accepts the Dwarf_Section object.

unit_hdr_offset
Input. This accepts a unit-header section offset.

ret_rootdie
Output. This returns a root DIE object.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_rootof operation returns DW_DLV_NO_ENTRY if the debug section is empty.

Example: Parameter deallocation

You can deallocate the parameters as required.

The following code fragment deallocates the ret_rootdie parameter:

```
if (dwarf_rootof (section,...&ret_rootdie, &err)
  == DW_DLV_OK) {
  dwarf_dealloc (ret_rootdie, DW_DLA_DIE);
}
```

Note: To simplify the example, only the relevant parameters are found in the above code. Unlisted parameters are represented by ellipses (...).

dwarf_parent operation

The dwarf_parent operation locates the parent DIE of a given DIE.

Prototype

```
int dwarf_parent(
  Dwarf_Die             die,
  Dwarf_Die*            ret_parentdie,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer object.

ret_parentdie
Output. This returns the parent DIE object.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_parent operation returns DW_DLV_NO_ENTRY if the given DIE does not have a parent.

Example: Parameter deallocation

You can deallocate the parameters as required.

The following code fragment deallocates the ret_parentdie parameter:

```
if (dwarf_parent (dbg, &ret_parentdie, &err)
  == DW_DLV_OK) {
  dwarf_dealloc (ret_parentdie, DW_DLA_DIE);
}
```

dwarf_offdie_in_section operation

The dwarf_offdie_in_section operation locates the DIE for a given section and offset.

Prototype

```
int dwarf_offdie_in_section(
  Dwarf_Section       section,
  Dwarf_Off           offset,
  Dwarf_Die*          ret_die,
  Dwarf_Error*        error);
```

Parameters

section
Input. This accepts the Dwarf_Section object.

offset
Input. This accepts a section offset.

ret_die
Output. This returns a DIE object.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_offdie_in_section operation returns DW_DLV_NO_ENTRY if the offset value is out of range.

Example: Parameter deallocation

You can deallocate the parameters as required.

The following code fragment deallocates the ret_die parameter:

```
if (dwarf_offdie_in_section (section,...&ret_die, &err)
  == DW_DLV_OK) {
  dwarf_dealloc (ret_die, DW_DLA_DIE);
}
```

Note: To simplify the example, only the relevant parameters are found in the above code. Unlisted parameters are represented by ellipses(...).

dwarf_nthdie operation

Given a DIE-format section unit, the dwarf_nthdie operation return a DIE given a DIE index. Every DIE has a unique DIE index, returned by dwarf_dieindex(). The upper limit of DIE index is given by dwarf_diecount()-1. Note that not every DIE index within this range maps to a DIE.

Prototype

```
int dwarf_nthdie(
  Dwarf_Section      section,
  Dwarf_Off          unit_hdr_offset
  Dwarf_Unsigned     die_index,
  Dwarf_Die*         ret_die,
  Dwarf_Error*       error);
```

Parameters

section
Input. This accepts the Dwarf_Section object.

unit_hdr_offset
Input. This accepts an offset for a unit-header section.

die_index
Input. This accepts a DIE index. Note that the root index value is 0.

ret_die
Output. This returns a DIE object.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_nthdie operation returns DW_DLV_NO_ENTRY if the die_index value is out of range.

Example: parameter deallocation

You can deallocate the parameters as required.

The following code fragment deallocates the ret_die parameter:

```
if (dwarf_nthdie (section,...&ret_die, &err)
  == DW_DLV_OK) {
  dwarf_dealloc (ret_die, DW_DLA_DIE);
}
```

Note: To simplify the example, only the relevant parameters are found in the above code. Unlisted parameters are represented by ellipses(...).

dwarf_clone operation

The dwarf_clone operation returns a copy of the Dwarf_Die object for the given DIE.

Prototype

```
int dwarf_clone(
  Dwarf_Die             die,
  Dwarf_Die*            ret_die,
  Dwarf_Error*          error);
```

Parameters

die
Input. This accepts the DIE object.

ret_die
Output. This returns the cloned DIE object.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_clone operation returns DW_DLV_NO_ENTRY if die is a NULL DIE (used to identify a DIE with no children).

Example: Parameter deallocation

You can deallocate the parameters as required.

Example: The code fragment deallocates the `ret_die` parameter:

```
if (dwarf_clone (die, &ret_die, &err)
  == DW_DLV_OK) {
  dwarf_dealloc (ret_die, DW_DLA_DIE);
}
```

dwarf_pcfile operation

The `dwarf_pcfile` operation returns the CU DIE that encloses a given PC address. A CU DIE is a DIE with a `DW_TAG_compile_unit` tag.

Prototype

```
int dwarf_pcfile(
  Dwarf_Debug           dbg,
  Dwarf_Addr            pc,
  Dwarf_Die*            ret_die,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a `libdwarf` consumer object.

pc
Input. This accepts the PC address.

ret_die
Output. This returns the DIE with a `DW_TAG_compile_unit` tag.

error
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

The `dwarf_pcfile` operation returns `DW_DLV_NO_ENTRY` if the `ret_die` does not contain the PC address.

Example: parameter deallocation

You can deallocate the parameters as required.

The following code fragment deallocates the `ret_die` parameter:

```
if (dwarf_pcfile (dbg, pc, &ret_die, &err) == DW_DLV_OK)
    dwarf_dealloc(dbg, ret_die, DW_DLA_DIE);
```

dwarf_pcsubr operation

The `dwarf_pcsubr` operation returns the subroutine DIE that encloses the given PC address.

A subroutine DIE is a DIE with a `DW_TAG_subprogram` tag.

Prototype

```
int dwarf_pcsubr(
  Dwarf_Debug           dbg,
  Dwarf_Addr            pc,
```

```
  Dwarf_Die*            ret_die,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a `libdwarf` consumer object.

pc
Input. This accepts the PC address.

ret_die
Output. This returns the DIE with a `DW_TAG_subprogram` tag.

error
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

The `dwarf_pcsubr` operation returns `DW_DLV_NO_ENTRY` if the `ret_die` does not contain the PC address.

Example: Parameter deallocation

You can deallocate the parameters as required.

Example: The following code fragment deallocates the `ret_die` parameter:

```
if (dwarf_pcsubr (dbg, pc, &ret_die, &err) == DW_DLV_OK)
    dwarf_dealloc(dbg, ret_die, DW_DLA_DIE);
```

dwarf_pcscope operation

The `dwarf_pcscope` operation returns the block DIE that encloses the given PC address with the smallest range.

The block DIE has a `DW_TAG_lexical_block` tag.

Prototype

```
int dwarf_pcscope(
  Dwarf_Debug           dbg,
  Dwarf_Addr            pc,
  Dwarf_Die*            ret_die,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a `libdwarf` consumer object.

pc
Input. This accepts the PC address.

ret_die
Output. This returns the block DIE that is closest to the given address.

error
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

The `dwarf_pcscope` operation returns `DW_DLV_NO_ENTRY` if the `ret_die` does not contain the PC address.

Example: Parameter deallocation

You can deallocate the parameters as required.

The following code fragment deallocates the `ret_die` parameter:

```
if (dwarf_pcscope (dbg, pc, &ret_die, &err) == DW_DLV_OK)
    dwarf_dealloc(dbg, ret_die, DW_DLA_DIE);
```

Multiple DIEs locating consumer operations

This section contains a list of APIs for locating a list of DIEs given one or more search criteria in s DIE section.

dwarf_tagdies operation

The `dwarf_tagdies` operation returns all of the DIEs in a given debug-section unit that have the specified tag.

Prototype

```
int dwarf_tagdies(
  Dwarf_Section      section,
  Dwarf_Off          unit_hdr_offset,
  Dwarf_Tag          tag,
  Dwarf_Die**        ret_dielist,
  Dwarf_Signed*      ret_diecount,
  Dwarf_Error*       error);
```

Parameters

`section`
Input. This accepts a `Dwarf_Section` object.

`unit_hdr_offset`
Input. This accepts a unit header section offset.

`tag`
Input. This accepts a DIE tag.

`ret_dielist`
Output. This returns a list of DIEs.

`ret_diecount`
Output. This returns a count of the DIEs in the list.

`error`
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

The `dwarf_tagdies` operation returns `DW_DLV_NO_ENTRY` if the given tag is not found in the given section.

Example: Parameter deallocation

You can deallocate the parameters as required.

The following code fragment deallocates the `ret_dielist` parameter:

```
if (dwarf_tagdies (section,...&ret_dielist, &ret_diecount, &err)
  == DW_DLV_OK) {
  for (i=0; i<diecount; i++)
    dwarf_dealloc (ret_dielist [i], DW_DLA_DIE);
```

```
  dwarf_dealloc (ret_dielist, DW_DLA_LIST);
}
```

Note: To simplify the example, only the relevant parameters are found in the above code. Unlisted parameters are represented by ellipses(...).

dwarf_attrdies operation

The `dwarf_attrdies` operation returns all the DIEs in a given debug-section unit that have a specified attribute.

Prototype

```
int dwarf_attrdies(
  Dwarf_Section       section,
  Dwarf_Off           unit_hdr_offset,
  Dwarf_Half          attr,
  Dwarf_Die**         ret_dielist,
  Dwarf_Signed*       ret_diecount,
  Dwarf_Error*        error);
```

Parameters

`section`
Input. This accepts a `Dwarf_Section` object.

`unit_hdr_offset`
Input. This accepts a unit header section offset.

`attr`
Input. This accepts the ID for a DIE attribute.

`ret_dielist`
Output. This returns a list of DIEs.

`ret_diecount`
Output. This returns a count of the DIEs in the list.

`error`
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

The `dwarf_attrdies` operation returns `DW_DLV_NO_ENTRY` if the `attr` value is not found in the given `section`.

Example: Parameter deallocation

You can deallocate the parameters as required.

Example: The following code fragment deallocates the `ret_dielist` parameter:

```
if (dwarf_tagdies (section,...&ret_dielist, &ret_diecount, &err)
  == DW_DLV_OK) {
  for (i=0; i<diecount; i++)
    dwarf_dealloc (ret_dielist [i], DW_DLA_DIE);
  dwarf_dealloc (ret_dielist, DW_DLA_LIST);
}
```

Note: To simplify the example, only the relevant parameters are found in the above code. Unlisted parameters are represented by ellipses(...).

dwarf_get_dies_given_name operation

The dwarf_get_dies_given_name operation returns a list of DIEs from a given section, whose DW_AT_name attributes match a given name.

Prototype

```
int dwarf_get_dies_given_name(
  Dwarf_Section         section,
  const char*           id_name,
  Dwarf_Die**           ret_dielist,
  Dwarf_Signed*         ret_diecount,
  Dwarf_Error*          error);
```

Parameters

section
Input. This accepts the Dwarf_Section object.

id_name
Input. This accepts the name to be compared with the DW_AT_name attribute of the DIEs in the section.

ret_dielist
Output. This returns a list of DIEs with a matching DW_AT_name attribute.

ret_diecount
Output. This returns the count of the DIEs in the list.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_get_dies_given_name operation returns DW_DLV_NO_ENTRY if none of the DW_AT_name attribute match id_name.

Example: Parameter deallocation

You can deallocate the parameters as required.

The following code fragment deallocates the ret_elf_symilst parameter:

```
if (dwarf_get_dies_given_name (section, id_name, &ret_dielist, &ret_diecount, &err)
  == DW_DLV_OK) {
  for (i=0; i<ret_diecount; i++)
    dwarf_dealloc (dbg, ret_dielist[i], DW_DLA_DIE);
  dwarf_dealloc (dbg, ret_dielist, DW_DLA_LIST);
}
```

dwarf_get_dies_given_pc operation

The dwarf_get_dies_given_pc operation returns a list of DIEs, from a given section, that enclose a given PC address.

The DIEs must have either DW_AT_low_pc and DW_AT_high_pc attributes, or a single DW_AT_range attribute. The dwarf_get_dies_given_pc operation reviews all the DIEs in the section and determines the low PC address and high PC address that is closest to the given address. It then returns all the DIEs with matching address attributes.

Prototype

```
int dwarf_get_dies_given_pc(
  Dwarf_Section         section,
```

```
  Dwarf_Addr            pcaddr,
  Dwarf_Die**           ret_dielist,
  Dwarf_Signed*         ret_diecount,
  Dwarf_Error*          error);;
```

Parameters

section
Input. This accepts the `Dwarf_Section` object.

pcaddr
Input. This accepts the initial PC address of the block.

ret_dielist
Output. This returns a list of DIEs that enclose the range.

ret_diecount
Output. This returns the count of the DIEs in the list.

error
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

The `dwarf_get_dies_given_pc` operation returns `DW_DLV_NO_ENTRY` if none of the DIEs contains the given PC address.

Example: Parameter deallocation

You can deallocate the parameters as required.

The following code fragment deallocates the `ret_dielist` parameter:

```
if (dwarf_get_dies_given_pc (section, pcaddr, &ret_dielist, &ret_diecount, &err)
  == DW_DLV_OK) {
  for (i=0; i<ret_diecount; i++)
    dwarf_dealloc (dbg, ret_dielist[i], DW_DLA_DIE);
  dwarf_dealloc (dbg, ret_dielist, DW_DLA_LIST);
}
```

DIE-query consumer operations

This section contains a list of APIs for specific information about a given DIE.

dwarf_diesection operation

The `dwarf_diesection` operation looks for the debug section and unit-header offset of a given DIE.

Prototype

```
int dwarf_diesection(
  Dwarf_Die           die,
  Dwarf_Section*      ret_section,
  Dwarf_Off*          ret_unit_hdrofs,
  Dwarf_Error*        error);
```

Parameters

die
Input. This accepts a DIE object.

ret_section
Output. This returns the `Dwarf_Section` object.

ret_unit_hdrofs
Output. This returns the section offset of the unit header.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_diesection operation never returns DW_DLV_NO_ENTRY.

dwarf_diecount operation

Given a DIE, the dwarf_diecount operation searches the containing DIE section and return number of DIE entries within the DIE section. For example, if this operation returns 12, then the allowable DIE index values are between 0 and 11.

Prototype

```
int dwarf_diecount(
  Dwarf_Die          die,
  Dwarf_Unsigned*    ret_die_count,
  Dwarf_Error*       error);
```

Parameters

die
Input. This accepts a DIE object.

ret_die_count
Output. This return the maximum DIE index + 1 for the DIE section. .

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_diecount operation never returns DW_DLV_NO_ENTRY.

dwarf_dieindex operation

The dwarf_dieindex operation returns the DIE index for a given DIE.

Prototype

```
int dwarf_dieindex(
  Dwarf_Die          die,
  Dwarf_Unsigned*    ret_die_index,
  Dwarf_Error*       error);
```

Parameters

die
Input. This accepts a DIE object.

ret_die_index
Output. This returns the DIE index. Please note that the root index value is 0.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_dieindex operation returns DW_DLV_NO_ENTRY if no index is found for die.

dwarf_isclone operation

The dwarf_isclone operation compares two Dwarf_Die objects to determine if they represent the same DIE.

Prototype

```
int dwarf_isclone(
  Dwarf_Die          die1,
  Dwarf_Die          die2,
  Dwarf_Bool*        returned_bool,
  Dwarf_Error*       error);
```

Parameters

die1
Input. This accepts the first DIE object.

die2
Input. This accepts the second DIE object.

returned_bool
Output. This returns the results of the test.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_isclone operation never returns DW_DLV_NO_ENTRY.

dwarf_dietype operation

The dwarf_dietype operation returns the DIE that is pointed to by the DW_AT_type attribute of a given DIE.

Prototype

```
int dwarf_dietype(
  Dwarf_Die             die,
  Dwarf_Die*            ret_typedie,
  Dwarf_Error*          error);
```

Parameters

die
Input. This accepts a DIE object with a DW_AT_type attribute.

ret_typedie
Output. This returns the DIE pointed to by the DW_AT_type attribute.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_dietype operation returns DW_DLV_NO_ENTRY if the die does not have a DW_AT_type attribute.

Example: Parameter deallocation

You can deallocate the parameters as required.

The following code fragment deallocates the `ret_die` parameter:

```
if (dwarf_pcscope (die, &ret_typedie, &err) == DW_DLV_OK)
    dwarf_dealloc(dbg, ret_typedie, DW_DLA_DIE);
```

dwarf_refdie operation

The `dwarf_refdie` operation returns the DIE that is pointed by an arbitrary attribute of a given DIE. The arbitrary attribute must be referencing a DIE within the same DWARF debug section, that is, the form of the attribute must be `DW_FORM_ref*` (where * can be 1, 2, 4, or 8), not `DW_FORM_ref_addr`.

Prototype

```
int dwarf_refdie(
    Dwarf_Die          die,
    Dwarf_Half         attr,
    Dwarf_Die*         ret_refdie,
    Dwarf_Error*       error);
```

Parameters

die
Input. This accepts a `Dwarf_Die` object with an attribute of form `DW_FORM_ref*`.

attr
Input. This is an attribute of form `DW_FORM_ref*`.

ret_refdie
Output. This returns the DIE that is pointed by `attr`.

error
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

The `dwarf_refdie` operation returns `DW_DLV_NO_ENTRY` if the given DIE does not have the user-specified attribute or the form of the attribute is not DW_FORM_ref*.

Example: Parameter deallocation

You can deallocate the parameters as required.

The following code fragment deallocates the `ret_refdie` parameter:

```
if (dwarf_refdie (die, attr, &ret_refdie, &err)
  == DW_DLV_OK) {
  dwarf_dealloc (dbg, ret_refdie, DW_DLA_DIE);
}
```

dwarf_refaddr_die operation

The `dwarf_refaddr_die` operation queries the DIE pointed by an arbitrary attribute. The arbitrary attribute can reference a DIE in any DWARF debug section. This API supports attribute form of `DW_FORM_refaddr` and `DW_FORM_sec_offset`.

Prototype

```
int dwarf_refaddr_die(
    Dwarf_Die           die,
```

```
    Dwarf_Half          attr,
    Dwarf_section_type  ref_sec_type,
    Dwarf_Die*          ret_refdie,
    Dwarf_Error*        error);
```

Parameters

die
Input. Input DIE object.

attr
Input. Input DIE attribute id that is referencing a DIE.

ref_sec_type
Input. DWARF section type of the referenced DIE.

ret_refdie
Output. Referenced DIE.

error
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

DW_DLV_OK
The DIE object referenced by the user specified attribute is returned.

DW_DLV_NO_ENTRY

- The given `die` does not have the user specified attribute.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- The given `die` is NULL.
- The given `ret_dies` is NULL.
- Cannot locate a DWARF debug instance associated with the given `die`.
- An error is encountered when allocating memory for the returned object.
- The form of the attribute and the given `ref_sec_type` is not a valid combination.

DIE-attribute query consumer operation

This section contains a list of APIs for querying a specific attribute about a given DIE.

dwarf_attr_offset operation

The `dwarf_attr_offset` operation returns the section offset of the given attribute.

Prototype

```
int dwarf_attr_offset(
  Dwarf_Die            die,
  Dwarf_Attribute     attr,
  Dwarf_Off*          attr_offset,
  Dwarf_Error*        error);
```

Parameters

die
Input. This accepts a DIE object.

attr
Input. This accepts a DIE attribute.

returned_offset
Output. This returns the offset of the attribute.

error
Input/output. This accepts or returns the Dwarf_Error object.

Note: This API relies on the input die to determine the DIE section that owns the attribute. If the die and the attr values are not related, the result is meaningless.

dwarf_data_bitoffset operation

The dwarf_data_bitoffset operation queries the bit offset attribute (DW_AT_data_bit_offset) associated with a given DIE.

Prototype

```
int dwarf_data_bitoffset(
    Dwarf_Die           die,
    Dwarf_Unsigned*     returned_offset,
    Dwarf_Error*        error);
```

Parameters

die
Input. This accepts a Dwarf_Die object.

returned_offset
Output. This returns the bit offset value in the DW_AT_data_bit_offset attribute.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_data_bitoffset operation returns DW_DLV_NO_ENTRY if DW_AT_data_bit_offset is not one of the attributes in die.

dwarf_die_xref_coord operation

The dwarf_die_xref_coord operation queries the DW_AT_IBM_xref_coord attribute associated with a given DIE. It retrieves the list of source coordinates in which the variable represented by the given DIE is referenced within the source program. The source coordinate is returned as a pair of integers (line number and column number).

Prototype

```
int
  dwarf_die_xref_coord(
    Dwarf_Die           die,
    Dwarf_Unsigned    **ret_lineno,
    Dwarf_Unsigned    **ret_colno,
    Dwarf_Unsigned     *ret_count,
    Dwarf_Error*        error);
```

Parameters

die
Input. This accepts a Dwarf_Die object.

ret_lineno
Output. This returns an array of elements containing the source line number.

ret_colno
Output. This returns an array of elements containing the source column number. This is zero if the column number is not used.

ret_count
Output. This returns the number of elements in the array.

error
Input/output. This accepts or returns the Dwarf_Error object.

Cleanups

```
Dwarf_Die       die;
Dwarf_Unsigned* lineno_arr;
Dwarf_Unsigned* colno_arr;
Dwarf_Unsigned  arr_count;

dwarf_die_xref_coord(die, &lineno_arr, &colno_arr, &arr_count, &err);

dwarf_dealloc (dbg, lineno_arr, DW_DLA_ADDR);
dwarf_dealloc (dbg, colno_arr, DW_DLA_ADDR);
```

Return values

DW_DLV_OK
DW_AT_IBM_xref_coord is found, and the list of source coordinates are returned.

DW_DLV_NO_ENTRY
DW_AT_IBM_xref_coord is not one of the attributes in the DIE.

DW_DLV_NO_ENTRY

DW_DLE_DIE_NULL
The given 'die' is NULL

DW_DLE_DBG_NULL
Can not locate a DWARF debug instance associated with the given 'die'

DW_DLE_RETURN_PTR_NULL
The given 'ret_lineno' or 'ret_colno' or 'ret_count' is NULL

DW_DLE_ALLOC_FAIL
There is an error allocating memory for the returned parameters.

High level PC location consumer APIs

These APIs support access to line-number programs and symbolic information for the instruction at a given PC location.

Dwarf_PC_Locn object

This opaque data type is used as a descriptor for queries about information related to a PC location. An instance of the Dwarf_PC_Locn type is created as a result of a successful call to dwarf_pclocns. The storage pointed to by this descriptor should be not be freed using the dwarf_dealloc operation. Instead free it with the dwarf_pc_locn_term operation.

Type definition

```
typedef struct Dwarf_PC_Locn_s* Dwarf_PC_Locn;
```

Dwarf_Subpgm_Locn object

This opaque data type is used as a descriptor for queries about subprogram line-number programs related to a PC location. An instance of the `Dwarf_Subpgm_Locn` type is created as a result of a successful call to the `dwarf_pc_locn_list` operation. This is a persistent copy and should not be freed.

Type definition

```
typedef struct Dwarf_Subpgm_Locn_s* Dwarf_Subpgm_Locn;
```

dwarf_pclocns operation

The `dwarf_pclocns` operation creates a PC object if given a PC address.

Prototype

```
int dwarf_pclocns(
  Dwarf_Debug         dbg,
  Dwarf_Addr          pc_of_interest,
  Dwarf_PC_Locn*      ret_locn,
  Dwarf_Error*        error);
```

Parameters

dbg
Input. This accepts a `libdwarf` consumer object.

pc_of_interest
Input. This accepts the PC address.

ret_locn
Output. This returns the `Dwarf_PC_Locn` object.

Refer to “Example: Parameter deallocation” on page 60.

error
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

The `dwarf_pclocns` operation returns DW_DLV_NO_ENTRY if the subprogram's line-number table does not exist.

Example: Parameter deallocation

You can deallocate the parameters as required.

The following code fragment deallocate the `ret_locn` parameter:

```
if (dwarf_pclocns (dbg,...&ret_locn, &err)
  == DW_DLV_OK) {
    dwarf_pc_locn_term (ret_locn, &err);
}
```

Note: For reasons of clarity, not all the parameters have been entered in the above code. Unlisted parameters are represented by ellipses (...).

dwarf_pc_locn_term operation

The dwarf_pc_locn_term operation terminates the given Dwarf_PC_Locn object.

Prototype

```
int dwarf_pc_locn_term(
  Dwarf_PC_Locn       locn,
  Dwarf_Error*        error);
```

Parameters

locn
Input. This accepts a Dwarf_PC_Locn object.

error
Input/output. This accepts or returns the Dwarf_Error object.

dwarf_pc_locn_abbr_name operation

The dwarf_pc_locn_abbr_name operation queries the abbreviated name for the given PC-location object.

Prototype

```
int dwarf_pc_locn_abbr_name(
  Dwarf_PC_Locn         locn,
  char**                ret_abbr_name,
  Dwarf_Error*          error);
```

Parameters

locn
Input. This accepts the Dwarf_PC_Locn object.

ret_abbr_name
Output. This returns the abbreviation for the name.

error
Input/output. This accepts or returns the Dwarf_Error object.

dwarf_pc_locn_set_abbr_name operation

The dwarf_pc_locn_set_abbr_name operation sets the abbreviated name for the given PC-location object.

Prototype

```
int dwarf_pc_locn_set_abbr_name(
  Dwarf_PC_Locn         locn,
  char*                 abbr_name,
  Dwarf_Error*          error);
```

Parameters

locn
Input. This accepts the Dwarf_PC_Locn object.

abbr_name
Input. This accepts the abbreviation name.

error
Input/output. This accepts or returns the Dwarf_Error object.

dwarf_pc_locn_entry operation

The dwarf_pc_locn_entry operation queries the entry information for a given Dwarf_PC_Locn object.

Prototype

```
int dwarf_pc_locn_entry(
  Dwarf_PC_Locn         locn,
  Dwarf_Die*            ret_unit_die,
  Dwarf_Off*            ret_ep_offset,
  Dwarf_Error*          error);
```

Parameters

locn
Input. This accepts the Dwarf_PC_Locn object.

ret_unit_die
Output. This returns the unit DIE.

ret_ep_offset
Output. This returns the entry point offset.

error
Input/output. This accepts or returns the Dwarf_Error object.

dwarf_pc_locn_list operation

The dwarf_pc_locn_list operation describes the subprograms which have contributed to a given PC object.

Prototype

```
int dwarf_pc_locn_list(
  Dwarf_PC_Locn         locn,
  Dwarf_Subpgm_Locn**   ret_subpgms,
  Dwarf_Signed*         ret_n_subpgms,
  Dwarf_Error*          error);
```

Parameters

locn
Input. This accepts the Dwarf_PC_Locn object.

ret_subpgms
Output. This returns the Dwarf_Subpgm_Locn object.

ret_n_subpgms
Output. This returns a count of the list entries.

error
Input/output. This accepts or returns the Dwarf_Error object.

dwarf_subpgm_locn operation

The dwarf_subpgm_locn operation queries the details from a subprogram contribution to a given PC address.

Prototype

```
int dwarf_subpgm_locn(
  Dwarf_Subpgm_Locn     subpgm_locn,
  Dwarf_Locn_Origin_t*  ret_origin,
  Dwarf_Die*            ret_subpgm_die,
  Dwarf_Line*           ret_line,
  Dwarf_Error*          error);
```

Parameters

subpgm_locn
Input. This accepts the Dwarf_Subpgm_Locn object.

ret_origin
Output. This returns the contribution type.

ret_subpgm_die
Output. This returns the subprogram DIE.

ret_line
Output. This returns the line-matrix row.

error
Input/output. This accepts or returns the Dwarf_Error object.

DWARF flag operations

This section contains a list of APIs for testing or setting the flag bits within a DWARF flag object.

dwarf_flag_any_set operation

The dwarf_flag_any_set operation tests whether or not any of the Dwarf_Flag index bit are set.

Prototype

```
int dwarf_flag_any_set (
  Dwarf_Debug           dbg,
  Dwarf_Flag*           flags,
  Dwarf_Bool*           ret_anyset,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer object.

flags
Input/Output. This accepts or returns the Dwarf_Flag object.

ret_anyset
Output. This returns the Boolean value which indicates whether or not any bit index is set.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_flag_any_set operation never returns DW_DLV_NO_ENTRY.

Memory deallocation

There is no storage to deallocate.

dwarf_flag_clear operation

The dwarf_flag_clear operation clears the given Dwarf_Flag index bit.

Prototype

```
int dwarf_flag_clear (
  Dwarf_Debug           dbg,
  Dwarf_Flag*           flags,
  int                   bit_idx,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer object.

flags
Input/Output. This accepts or returns the Dwarf_Flag object.

bit_idx
Input. This accepts the flag bit index to clear. It can be a value from 0 to 31.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_flag_clear operation never returns DW_DLV_NO_ENTRY.

Memory deallocation

There is no storage to deallocate.

dwarf_flag_complement operation

The dwarf_flag_complement operation complements the given Dwarf_Flag index bit.

Prototype

```
int dwarf_flag_complement (
  Dwarf_Debug           dbg,
  Dwarf_Flag*           flags,
  int                   bit_idx,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer object.

flags
Input/Output. This accepts or returns the Dwarf_Flag object.

bit_idx
Input. This accepts the flag bit index to complement. It can be a value from 0 to 31.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_flag_complement operation never returns DW_DLV_NO_ENTRY.

Memory allocation

There is no storage to deallocate.

dwarf_flag_copy operation

The dwarf_flag_copy operation sets or clears the given Dwarf_Flag bit index.

dwarf_flag_copy copies a given Boolean value into the bit index.

Prototype

```
int dwarf_flag_copy (
  Dwarf_Debug           dbg,
  Dwarf_Flag*           flags,
  int                   bit_idx,
  Dwarf_Bool            val,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer object.

flags
Input/Output. This accepts or returns the Dwarf_Flag object.

bit_idx
Input. This accepts the flag bit index to set or clear. It can be a value from 0 to 31.

val
Input. This accepts the Boolean value which indicates whether to set or clear the bit index.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_flag_copy operation never returns DW_DLV_NO_ENTRY.

Memory deallocation

There is no storage to deallocate.

dwarf_flag_reset operation

The dwarf_flag_reset operation clears all the Dwarf_Flag index bits.

Prototype

```
int dwarf_flag_reset (
  Dwarf_Debug           dbg,
  Dwarf_Flag*           flags,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer object.

flags
Input/Output. This accepts or returns the Dwarf_Flag object.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_flag_reset operation never returns DW_DLV_NO_ENTRY.

Memory deallocation

There is no storage to deallocate.

dwarf_flag_set operation

The dwarf_flag_set operation sets the given Dwarf_Flag index bit.

Prototype

```
int dwarf_flag_set (
  Dwarf_Debug           dbg,
  Dwarf_Flag*           flags,
  int                   bit_idx,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer object.

flags
Input/Output. This accepts or returns the Dwarf_Flag object.

bit_idx
Input. This accepts the flag bit index to set. It can be a value from 0 to 31.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_flag_set operation never returns DW_DLV_NO_ENTRY.

Memory deallocation

There is no storage to deallocate.

dwarf_flag_test operation

The dwarf_flag_test operation tests whether or not the given Dwarf_Flag index bit is set.

Prototype

```
int dwarf_flag_test (
  Dwarf_Debug           dbg,
  Dwarf_Flag*           flags,
  int                   bit_idx,
  Dwarf_Bool*           ret_bitset,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a `libdwarf` consumer object.

flags
Input/Output. This accepts or returns the `Dwarf_Flag` object.

bit_idx
Input. This accepts the flag bit index to test. It can be a value from 0 to 31.

ret_bitset
Output. This returns the Boolean value which indicates whether or not the bit index is set.

error
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

The `dwarf_flag_test` operation never returns `DW_DLV_NO_ENTRY`.

Memory deallocation

There is no storage to deallocate.

Accelerated access consumer operations

This section contains a list of APIs related to accelerated access debug sections. For more information about accelerated access debug sections, refer to Section 6.1 in *DWARF Debugging Information Format*, V4.

For a description of DWARF debugging sections, see “Dwarf_section_type enumeration” on page 40.

IBM extensions to accelerated access debug sections

This section provides a list of accelerated access debug sections supported by CDA.

Lookup by Name debug sections available via standard DWARF:

.debug_pubnames
Stores names of global objects and functions.

.debug_pubtypes
Stores names of global types.

.debug_funcnames
Stores names of file-scoped static functions.

.debug_varnames
Stores names of file-scoped static data symbols.

.debug_weaknames
Stores names of weak symbols.

Lookup by Address debug sections available via standard DWARF:

.debug_aranges
Stores addresses of compilation units.

Dwarf_section_type object

The Dwarf_section_type data structure allows access to the ELF information through the DWARF sections. `Dwarf_section_type` can access section numbers and ELF section name indexes in the symbol table.

Type definition

```
typedef enum Dwarf_section_type_s {
  DW_SECTION_DEBUG_INFO        =  0,
  DW_SECTION_DEBUG_LINE        =  1,
  DW_SECTION_DEBUG_ABBREV      =  2,
  DW_SECTION_DEBUG_FRAME       =  3,
  DW_SECTION_EH_FRAME          =  4,
  DW_SECTION_DEBUG_ARANGES     =  5,
  DW_SECTION_DEBUG_RANGES      =  6,
  DW_SECTION_DEBUG_PUBNAMES    =  7,
  DW_SECTION_DEBUG_PUBTYPES    =  8,
  DW_SECTION_DEBUG_STR         =  9,
  DW_SECTION_DEBUG_FUNCNAMES   =  10,
  DW_SECTION_DEBUG_VARNAMES    =  11,
  DW_SECTION_DEBUG_WEAKNAMES   =  12,
  DW_SECTION_DEBUG_MACINFO     =  13,
  DW_SECTION_DEBUG_LOC         =  14,
  DW_SECTION_DEBUG_PPA         =  15,
  DW_SECTION_DEBUG_SRCFILES    =  16,
  DW_SECTION_DEBUG_SRCTEXT     =  17,
  DW_SECTION_DEBUG_SRCATTR     =  18,
  DW_SECTION_DEBUG_XREF        =  19,
  DW_SECTION_NUM_SECTIONS
} Dwarf_section_type;
```

Only the following DWARF sections are accelerated access debug sections, and can be used for accelerated access debug section APIs:

```
DW_SECTION_DEBUG_ARANGES
DW_SECTION_DEBUG_PUBNAMES
DW_SECTION_DEBUG_PUBTYPES
DW_SECTION_DEBUG_FUNCNAMES
DW_SECTION_DEBUG_VARNAMES
DW_SECTION_DEBUG_WEAKNAMES
```

Members

```
Dwarf_section_type           ELF section name   GOFF class name
=============================================================
DW_SECTION_DEBUG_INFO        .debug_info        D_INFO
DW_SECTION_DEBUG_LINE        .debug_line        D_LINE
DW_SECTION_DEBUG_ABBREV      .debug_abbrev      D_ABREV
DW_SECTION_DEBUG_FRAME       .debug_frame       D_FRAME
DW_SECTION_EH_FRAME          .eh_frame          n/a
DW_SECTION_DEBUG_ARANGES     .debug_aranges     D_ARNGE
DW_SECTION_DEBUG_RANGES      .debug_ranges      D_RNGES
DW_SECTION_DEBUG_PUBNAMES    .debug_pubnames    D_PBNMS
DW_SECTION_DEBUG_PUBTYPES    .debug_pubtypes    D_TYPES
DW_SECTION_DEBUG_STR         .debug_str         D_STR
DW_SECTION_DEBUG_FUNCNAMES   .debug_funcnames   D_SFUNC
DW_SECTION_DEBUG_VARNAMES    .debug_varnames    D_SVAR
DW_SECTION_DEBUG_WEAKNAMES   .debug_weaknames   D_WEAK
DW_SECTION_DEBUG_MACINFO     .debug_macinfo     D_MACIN
DW_SECTION_DEBUG_LOC         .debug_loc         D_LOC
DW_SECTION_DEBUG_PPA         .debug_ppa         D_PPA
DW_SECTION_DEBUG_SRCFILES    .debug_srcfiles    D_SRCF
DW_SECTION_DEBUG_SRCTEXT     .debug_srctext     D_SRCTXT
DW_SECTION_DEBUG_SRCATTR     .debug_srcattr     D_SRCATR
DW_SECTION_DEBUG_XREF        .debug_xref        D_XREF
```

dwarf_access_aranges operation

The dwarf_access_aranges operation returns all the address-range information for a given consumer object, in ascending order by address.

Prototype

```
int dwarf_access_aranges(
  Dwarf_Debug           dbg,
  Dwarf_Arange**        aranges,
  Dwarf_Signed*         arange_count,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer object.

aranges
Output. This returns the list of Dwarf_Arange entries.

highpc
Output. This returns the count of entries in the list.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_access_aranges operation never returns DW_DLV_NO_ENTRY.

Memory allocation

The address range array is a persistent copy, associated with the consumer instance. The array must be deallocated by dwarf_finish.

dwarf_find_arange operation

The dwarf_find_arange operation uses a binary search and returns the address-range entry for a given PC location.

Prototype

```
int dwarf_find_arange (
  Dwarf_Debug           dbg,
  Dwarf_Addr            pc_of_interest,
  Dwarf_Arange*         returned_arange,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer object.

pc_of_interest
Input. This accepts a PC address.

returned_arange
Output. This returns the address-range entry for the PC address.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_find_arange operation never returns DW_DLV_NO_ENTRY.

Memory allocation

There is no storage to deallocate.

dwarf_get_die_given_name_cuoffset operation

The dwarf_get_die_given_name_cuoffset operation queries a global name lookup table, searching for a DIEs that match a given a name.

The search is narrowed by specifying the required unit-header offsets. This function can find a single, specific match, if it exists in the DWARF file.

Prototype

```
int dwarf_get_die_given_name_cuoffset (
  Dwarf_Debug           dbg,
  Dwarf_section_type    sec_type,
  const char*           name,
  Dwarf_Off             unit_hdr_off,
  Dwarf_Die**           ret_die,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer object.

sec_type
Input. This accepts the name of the debug section containing the name lookup table.

name
Input. This accepts the name.

unit_hdr_off
Input. This accepts the unit-header offset.

ret_die
Output. This returns the DIE object.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

If the value of the name parameter cannot be found in the specified lookup table, DW_DLV_NO_ENTRY is returned.

Memory allocation

You can deallocate the parameters as required.

Example: The following example is a code fragment that deallocates the ret_die parameter:

```
if (dwarf_get_die_given_name_cuoffset (dbg,...&ret_die, &err)
  == DW_DLV_OK) {
  dwarf_dealloc (dbg, ret_die, DW_DLA_DIE);
}
```

Note: For reasons of clarity, not all the parameters have been entered in the above code. Unlisted parameters are represented by ellipses (...).

dwarf_get_dies_given_nametbl operation

The `dwarf_get_dies_given_nametbl` operation queries a global name lookup table, searching for DIEs with a given a name.

The search is narrowed to sections with a given section name.

Prototype

```
int dwarf_get_dies_given_nametbl (
  Dwarf_Debug           dbg,
  Dwarf_section_type    sec_type,
  const char*           name,
  Dwarf_Die**           ret_dielist,
  Dwarf_Unsigned*       ret_diecount,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a `libdwarf` consumer object.

sec_type
Input. This accepts one of the five valid types for the name lookup table.

name
Input. This accepts the name of an entry within the lookup table.

ret_dielist
Output. This returns a list of DIE objects.

ret_diecount
Output. This returns the count of the DIE objects in the list.

error
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

If the debug sections for the name lookup table have multiple entries with the same name, then all entries matching the name will be returned. If the value of the `name` parameter cannot be found in the specified lookup table, then `DW_DLV_NO_ENTRY` is returned.

Memory allocation

You can deallocate the parameters as required.

Example: The following example is a code fragment that deallocates the `dielist` parameter:

```
if (dwarf_get_dies_given_nametbl (dbg,...&dielist, &diecount, &err)
  == DW_DLV_OK) {
  for (i=0; i<diecount; i++)
    dwarf_dealloc (dbg, dielist[i], DW_DLA_DIE);
  dwarf_dealloc (dbg, dielist, DW_DLA_LIST);
}
```

Note: For reasons of clarity, not all the parameters have been entered in the above code. Unlisted parameters are represented by ellipses (...).

Non-contiguous address ranges consumer operations

This sections contains a list of APIs for querying information within the `.debug_ranges` section.

dwarf_get_ranges_given_offset operation

The dwarf_get_ranges_given_offset operation returns a unordered list of address ranges for given an offset within the .debug_ranges section.

Prototype

```
int dwarf_get_ranges_given_offset (
  Dwarf_Debug           dbg,
  Dwarf_Off             offset,
  Dwarf_Ranges**        ret_ranges,
  Dwarf_Unsigned*       ret_count,
  Dwarf_Off*            ret_nextoff,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer object.

offset
Input. This accepts the offset to use in the .debug_ranges section.

ret_ranges
Output. This returns the array of ranges.

ret_count
Output. This returns the number of entries in the array.

ret_nextoff
Output. This returns the offset of the next entry in the array.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

dwarf_get_ranges_given_offset returns DW_DLV_NO_ENTRY if either the .debug_info or the .debug_ranges section is empty.

Memory allocation

You can deallocate the parameters as required.

Example: The following example is a code fragment that deallocates the ret_ranges parameter:

```
if (dwarf_get_ranges_given_offset (dbg,...&ret_ranges, &ret_count,...&err)
  == DW_DLV_OK) {
  for (i=0; i<ret_count; i++)
    dwarf_dealloc (dbg, ret_ranges[i], DW_DLA_RANGES);
  dwarf_dealloc (dbg, ret_ranges, DW_DLA_LIST);
}
```

Note: For reasons of clarity, not all the parameters have been entered in the above code. Unlisted parameters are represented by ellipses (...).

dwarf_range_highpc operation

The dwarf_range_highpc operation returns the high PC of a given range entry.

Prototype

```
int dwarf_range_highpc (
  Dwarf_Debug           dbg,
  Dwarf_Ranges          range_entry,
```

```
  Dwarf_Addr*           highpc,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer object.

range_entry
Input. This accepts the range entry.

highpc
Output. This returns the high PC of the range entry.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

dwarf_range_highpc returns DW_DLV_NO_ENTRY if the range entry is empty.

Memory allocation

There is no storage to deallocate.

dwarf_range_lowpc operation

The dwarf_range_lowpc operation returns the low PC of a given range entry.

Prototype

```
int dwarf_range_lowpc (
  Dwarf_Debug           dbg,
  Dwarf_Ranges          range_entry,
  Dwarf_Addr*           lowpc,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer object.

range_entry
Input. This accepts the range entry.

lowpc
Output. This returns the low PC of the range entry.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

dwarf_range_lowpc returns DW_DLV_NO_ENTRY if the range entry is empty.

Memory allocation

There is no storage to deallocate.

Chapter 4. Program Prolog Area (PPA) extension

The Program Prolog Area (PPA) blocks are data areas in DWARF consumer APIs that conform to the Language Environment runtime conventions.

PPA blocks are generated by a language translator, which might be either of:

- A compiler.
- A high-level assembler (HLASM), when using the appropriate LE prolog and epilog macros.

PPA blocks are also referred to as Prolog Information Blocks.

An application can use the PPA blocks to:

- Identify compilation units (CUs) and some of their characteristics (PPA2).
- Identify subprograms (that is, functions, methods, subroutines) and some of their characteristics (PPA1).

IBM has created extensions to the DWARF sections and Debug Information Entries (DIEs) to support PPA information. For more information about these sections, refer to Appendix 7 in DWARF Debugging Information Format, V3, Draft 7.

Debug section

This section discusses the PPA debug section, which is an IBM extension.

The .debug_ppa section is an IBM extension. It contains Debug Information Entries (DIEs) which describe the PPA blocks in each application executable module. The PPA block information is used to permit a common set of high-level routines to provide access to the program attribute information which is stored in, or located by, each PPA block. This information originates during the program translation process (compilation or assembly), and initially describes the PPA blocks for a single CU.

The .debug_ppa section is required when relocating the ELF file. The relocation process is as follows:

- A scan of the module storage is performed to locate each PPA1 and PPA2 block
- The location of each PPA block is determined
- The location of all .debug_ppa sections are adjusted to match the physical location of each PPA block in the module

The granularity of the .debug_ppa information is at the CU level. A separate block will be generated that contains the DIEs for a single PPA2 block and the associated set of PPA1 blocks. Each .debug_ppa section block may share the associated .debug_abbr section block, but will have a separate .rela.debug_ppa relocation section block.

The following example shows a typical .debug_ppa section:

```
.debug_ppa

<header overall offset = 0>unit_hdr_off:
<0><   11>      DW_TAG_IBM_ppa2
                DW_AT_low_pc                0x108
                DW_AT_IBM_ppa_owner         11
                DW_AT_name                  .ppa2_b_B078078AFCCD2F705FDE73A5D3D4E967
<1><   61>      DW_TAG_IBM_ppa1
                DW_AT_low_pc                0x98
                DW_AT_IBM_ppa_owner         322
```

For more information about the structure of debug sections, see DWARF program information.

Block header

Each block of information in the .debug_ppa section begins with a header that contains the location-format information. This header does not replace any debugging information entries. It is additional information that is represented outside the standard DWARF tag/attributes format. It is used to navigate the information blocks in the .debug_ppa section. This is similar in format and intent to the standard Compile-Unit Header

The .debug_ppa block header contains:

1. block_length (initial length). A 4-byte or 12-byte unsigned integer representing the length of the .debug_ppa block, not including the length of the field itself. In the 32-bit Dwarf format, this is a 4-byte unsigned integer (which must be less than 0xFFFFFF00). In the 64-bit format, this is a 12-byte unsigned integer that consists of the 4-byte value 0xFFFFFFFF followed by an 8-byte unsigned integer that gives the actual value of the integer.
2. version. A 2-byte unsigned integer representing the version of the DWARF information for that block of .debug_ppa information
3. debug_abbrev_offset (section offset). A 4-byte or 8-byte unsigned offset into the .debug_abbrev section that associates the PPA location format information with a particular set of debugging information entry abbreviations
4. address_size (ubyte). A 1-byte unsigned integer representing the size in bytes of an address on the target architecture. If the system uses segmented addressing, this value represents the size of the offset portion of an address.

Section-specific DIEs

A .debug_ppa section can have the following DIEs:

- DW_TAG_IBM_ppa1 describes a single PPA1 block. It can be a child of a DW_TAG_IBM_ppa2 DIE.
- DW_TAG_IBM_ppa2 describes a single PPA2 block and its related set of CU-level PPA1 location information.

Reference section

DIEs in the .debug_ppa block can reference the following DIEs:

- Other DIEs in the .debug_ppa section
- DIEs in the .debug_info section.

A PPA2 (CU-level) block:

- Is described by a DW_TAG_IBM_ppa2 DIE
- Can contain a DW_AT_low_pc attribute to describe the starting address of the block
- Can contain a DW_AT_IBM_ppa_owner attribute to describe the location of the corresponding DW_TAG_compilation_unit DIE in the .debug_info section
- Can contain a DW_AT_name attribute to describe a unique signature to identify the CU

A PPA1 block:

- Is described by a DW_TAG_IBM_ppa1 DIE, using a DW_AT_low_pc attribute
- Can contain a DW_AT_low_pc attribute to describe the starting address of the block
- Can contain a DW_AT_IBM_ppa_owner attribute to describe the location of the corresponding DW_TAG_subprogram DIE in the .debug_info section

Companion sections

For each block of information in the .debug_ppa block, there will also be an associated block in the .debug_abbrev and .rela.debug_ppa sections.

.debug_abbrev contains a list of abbreviation tables. The tables describe the low-level encoding for each particular form of DIE. This will be a DIE tag, optionally associated with a specific grouping of attribute entries. Each attribute will have an associated form code which describes the precise encoding of the data for each attribute. For more information about abbreviation-table encoding, see the DWARF Debugging Information Format Standard, V3, Draft 7.

.rel.debug_ppa contains ELF-format relocation entries which are used to perform relocations related to the .debug_ppa information. These relocations are section offsets only.

While not strictly part of the .debug_ppa information, there are additional blocks of debug sections that would also normally be generated to make this section useful. These include the .debug_info and .debug_line sections.

Attributes forms

The DWARF attribute form governs how the value of a Debug Information Entry (DIE) attribute is encoded. The IBM extensions to DWARF do not introduce new attribute form codes, but extend their usage.

The Attribute Form Class `ppaptr` can identify any debugging information entry within a `.debug_ppa` section. This type of reference (`DW_FORM_sec_offset` in DWARF V4, `DW_FORM_data4` and `DW_FORM_data8` in DWARF V3) is an offset from the beginning of the `.debug_ppa` section.

PPA consumer operations

This section discusses the PPA consumer operations.

dwarf_get_all_ppa2dies operation

The `dwarf_get_all_ppa2dies` operation finds and returns the list of all `DW_TAG_IBM_ppa2` DIE objects.

Prototype

```
int dwarf_get_all_ppa2dies (
   Dwarf_Debug           dbg,
   Dwarf_Die**           ret_dielist,
   Dwarf_Signed*         ret_diecount,
   Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a `libdwarf` consumer object.

ret_dielist
Output. This returns a list of PPA2 DIE objects.

ret_diecount
Output. This returns the count of the PPA2 DIE objects in the list.

error
Input/output. This accepts and returns the `Dwarf_Error` object.

Return values

The `dwarf_get_all_ppa2dies` operation returns `DW_DLV_NO_ENTRY` if it cannot find any PPA2 DIE objects in the specified unit of the debug section.

Memory allocation

You can deallocate the parameters as required.

Example: A code fragment that deallocates the `ret_dielist` parameter:

```
if (dwarf_get_all_ppa2dies (dbg,&dielist, &diecount, &err)
  == DW_DLV_OK) {
  for (i=0; i < diecount; i++)
    dwarf_dealloc (dbg, dielist[i], DW_DLA_DIE);
  dwarf_dealloc (dbg, dielist, DW_DLA_LIST);
}
```

dwarf_get_all_ppa1dies_given_ppa2die operation

The `dwarf_get_all_ppa1dies_given_ppa2die` operation returns a list of `DW_TAG_IBM_ppa1` DIE objects for a given `DW_TAG_IBM_ppa2` DIE object.

Prototype

```
int dwarf_get_all_ppa1dies_given_ppa2die (
  Dwarf_Debug           dbg,
  Dwarf_Die             ppa2_die,
  Dwarf_Die**           ret_dielist,
  Dwarf_Signed*         ret_diecount,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a `libdwarf` consumer object.

ppa2_die
Input. This accepts a PPA2 DIE object.

ret_dielist
Output. This returns a list of PPA2 DIE objects.

ret_diecount
Output. This returns the count of the PPA2-DIE objects in the list.

error
Input/output. This accepts and returns the `Dwarf_Error` object.

Return values

The `dwarf_get_all_ppa1dies_given_ppa2die` operation returns `DW_DLV_NO_ENTRY` if it cannot find any PPA1 DIE objects in the specified debug-section unit.

Memory allocation

You can deallocate the parameters as required.

Example: A code fragment that deallocates the `ret_dielist` parameter:

```
if (dwarf_get_all_ppa1dies_given_ppa2die (dbg,...&dielist, &diecount, &err)
  == DW_DLV_OK) {
  for (i=0; i<diecount; i++)
    dwarf_dealloc (dbg, dielist[i], DW_DLA_DIE);
  dwarf_dealloc (dbg, dielist, DW_DLA_LIST);
}
```

Note: For reasons of clarity, not all the parameters have been entered in the above code. Unlisted parameters are represented by ellipses (...).

For more information about deallocating the `error` parameter, see *Consumer Library Interface to DWARF*, by the UNIX International Programming Languages Special Interest Group.

dwarf_get_all_ppa2die_given_cu_offset operation

The dwarf_get_all_ppa2die_given_cu_offset operation finds the DW_TAG_IBM_ppa2 DIE object for a given CU offset in the .debug_info section.

Prototype

```
int dwarf_get_ppa2die_given_cu_offset (
  Dwarf_Debug        dbg,
  Dwarf_Off          offset,
  Dwarf_Die*         ret_ppa2_die,
  Dwarf_Error*       error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer object.

offset
Input. This accepts the offset to be used within the .debug_info section.

ret_ppa2_die
Output. This returns the PPA2 DIE object.

error
Input/output. This accepts and returns the Dwarf_Error object.

Return values

The dwarf_get_all_ppa2die_given_cu_offset operation returns DW_DLV_NO_ENTRY if none of the PPA2 DIEs refer to the specified offset of the CU.

Memory allocation

You can deallocate the parameters as required.

Example: A code fragment that deallocates the ret_ppa2_die parameter:

```
if (dwarf_get_ppa2die_given_cu_offset (dbg, offset, &ret_ppa2_die, &err)
  == DW_DLV_OK) {
  dwarf_dealloc (dbg, ret_ppa2_die, DW_TAG_IBM_ppa2);
}
```

dwarf_find_ppa operation

The dwarf_find_ppa operation finds the PPA2 and PPA1 blocks associated with a given program-counter (PC) address and returns the PPA2 and PPA1 DIE objects.

Prototype

```
int dwarf_find_ppa(
  Dwarf_Debug       dbg,
  Dwarf_Addr        pc_of_interest,
  Dwarf_Addr*       ret_ppa2_addr,
  Dwarf_Die*        ret_ppa2_die,
  Dwarf_Die*        ret_root_die,
  Dwarf_Addr*       ret_ppa1_addr,
  Dwarf_Die*        ret_ppa1_die,
  Dwarf_Die*        ret_subr_die,
  Dwarf_Error*      error);
```

Parameters

dbg
Input. This accepts a `libdwarf` consumer object.

pc_of_interest
Input. This accepts the requested program-counter address.

ret_ppa2_addr
Output. This returns the PPA2 block address.

ret_ppa2_die
Output. This returns the PPA2 DIE object from the `.debug_ppa` section.

ret_root_die
Output. This returns the root DIE object from the `.debug_info` section.

ret_ppa1_addr
Output. This returns the PPA1 block address.

ret_ppa1_die
Output. This returns the PPA1 DIE object from the `.debug_ppa` section.

ret_subr_die
Output. This returns the subprogram DIE object from the `.debug_info` section.

error
Input/output. This accepts and returns the `Dwarf_Error` object.

Return values

The `dwarf_find_ppa` operation returns `DW_DLV_NO_ENTRY` if none of the PPA2 blocks are associated with the given `pc_of_interest`.

Memory allocation

You can deallocate the parameters as required.

Example: A code fragment that deallocates the `ret_ppa2_addr` parameter:

```
if (dwarf_find_ppa (dbg, pc_of_interest,
                        &ret_ppa2_addr,
                        &ret_ppa2_die,
                        &ret_root_die,
                        &ret_ppa1_addr,
                        &ret_ppa1_die,
                        &ret_subr_die,
                        &err) == DW_DLV_OK) {
    dwarf_dealloc(dbg, ret_ppa2_die, DW_DLA_DIE);
    dwarf_dealloc(dbg, ret_root_die, DW_DLA_DIE);
    dwarf_dealloc(dbg, ret_ppa1_die, DW_DLA_DIE);
    dwarf_dealloc(dbg, ret_subr_die, DW_DLA_DIE);
}
```

Chapter 5. Program source cross reference

This section contains debugging information entries that provide cross reference information between a program source file and debugging information entries that are contained in the `.debug_info` section.

Cross reference information for an object file may be contributed by one or more source file units. Each source file unit is represented by a cross reference unit debugging information entry with the tag `DW_TAG_IBM_xref_unit`.

Debug section

The `.debug_xref` section contains debugging information entries that provide cross reference information between a program source file and debugging information entries contained in the `.debug_info` section.

Block header

Each block of information in the `.debug_xref` section begins with a header that contains the location-format information. This header does not replace any debugging information entries. It is additional information that is represented outside the standard DWARF tag/attributes format. It is used to navigate the information blocks in the `.debug_xref` section. This is similar in format and intent to the standard Compile-Unit Header for `.debug_info`.

The `.debug_xref` block header contains:

Block length
A 4-byte or 12-byte unsigned integer representing the length of the `.debug_pa` block, not including the length of the field itself. In the 32-bit Dwarf format, this is a 4-byte unsigned integer (which must be less than 0xFFFFFF00). In the 64-bit format, this is a 12-byte unsigned integer that consists of the 4-byte value 0xFFFFFFFF followed by an 8-byte unsigned integer that gives the actual value of the integer.

DWARF version
A 2-byte unsigned integer representing the version of the DWARF information for that block of `.debug_xref` information.

.debug_abbrev offset
A 4-byte or 8-byte unsigned offset into the `.debug_abbrev` section that associates the `.debug_xref` information with a particular set of debugging information entry abbreviations.

Address size
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.

Section-specific DIEs

The debugging information entries contained in the `.debug_xref` section provide cross reference information between a program source file and debugging information entries contained in the `.debug_info` section.

Cross reference information for an object file may be contributed by one or more source file units. Each source file unit is represented by a cross reference unit debugging information entry with the tag `DW_TAG_IBM_xref_unit`.

A cross reference unit entry owns debugging information entries that represent all cross reference data within the source file unit. Cross reference unit entries may have the following attributes:

- a `DW_AT_IBM_src_file` attribute whose value is a reference. This attribute points to a debugging information entry within `.debug_srcfiles` containing detail information about the source file.

- a DW_AT_IBM_owner attribute whose value is a reference. This attribute points to a compilation unit debugging information entry which all cross reference DIE references belong to.

Each source line within a source file unit can have one or more statements. Each statement can have zero or more cross reference items. A statement containing cross reference items is represented by a debugging information entry with the tag DW_TAG_IBM_xreflist. The parent of the statement debugging information entry is the owning cross reference unit entry DIE. A statement DIE does not have any attribute, and it may have the following cross reference item(s) as children.

There are two types of cross reference items:

- A data variable referenced on a statement is represented by a debugging information entry with the tag DW_TAG_IBM_xreflist_item.
- A call to a subprogram/label on a statement is represented by a debugging information entry with the tag DW_TAG_IBM_on_call_item.

Each cross reference item debugging information entry may have the following attributes::

- a DW_AT_name attribute whose value is a string representing the name of cross reference item as it appears in the source program.
- a DW_AT_IBM_xreflist_item attribute whose value is a reference. This attribute points to a debugging information entry within .debug_info describing the declaration of the cross reference item.
- a DW_AT_IBM_is_modified attribute whose value is a flag. It is applicable to DW_TAG_IBM_xreflist_item only. This attribute indicates that the cross reference item is being modified by the statement.
- a DW_AT_IBM_call_type attribute whose integer constant value is a code describing the how the call is made. It is applicable to DW_TAG_IBM_on_call_item only. If the attribute is missing, the default value DW_CT_func_call is assumed. The set of call type codes is:

```
DW_CT_func_call    = 0,     /* Normal function call               */
DW_CT_alter          = 1,  /* ALTER <label&gt;                   */
DW_CT_alter_proceed  = 2,  /* ALTER ... TO PROCEED TO <label&gt; */
DW_CT_perform        = 3,  /* PERFORM <label&gt;                 */
DW_CT_perform_thru   = 4,  /* PERFORM ... THROUGH <label&gt;     */
DW_CT_goto           = 5,  /* GO TO <label&gt;                   */
DW_CT_goto_depend    = 6,  /* GO TO <label&gt; DEPENDING ON      */
DW_CT_use_for_debug  = 7   /* USE FOR DEBUGGING ON <label&gt;    */
```

Reference section

DIEs in the .debug_xref block can reference the following DIEs:

- DIEs in the .debug_xref section
- DIEs in the .debug_infosection

The following section can reference DIEs in the .debug_xref section:

- .debug_srcattr

Companion sections

For each block of information in the .debug_xref block, there will also be an associated block in the .debug_abbrev and .rel.debug_xref sections.

.debug_abbrev contains a list of abbreviation tables. The tables describe the low-level encoding for each particular form of DIE. This will be a DIE tag, optionally associated with a specific grouping of attribute entries. Each attribute will have an associated form code which describes the precise encoding of the data for each attribute. For more information about abbreviation-table encoding, see the *DWARF Debugging Information Format Standard*, V4.

`.rel.debug_xref` applies to ELF object file only and contain ELF-format relocation entries which are used to perform relocations related to the `.debug_xref` information. These relocations are section offsets only.

Chapter 6. Program line-number extensions

The DWARF standard defines the `.debug_line` section. This section contains a Line Number Program for each CU, which is encoded in a portable compact manner, for execution and expansion by the `libdwarf` Line Number Program state machine. This provides access to program source line and address information for the CU. CDA currently can consume and produce version 3 of the `.debug_line` section.

In the z/OS Common Debug Architecture, the following IBM extensions to this program are defined:

- Extensions relate to breakpoint type flags, and symbol declaration coordinates.
- Extensions relate to program source files and source text lines. Source file names and location information is moved from the CU-level Statement Program to the global `.debug_srcfiles` section.

Breakpoint type flags

Each standard DWARF line number program matrix row contains a given number of DWARF attribute flags. These are typically used to determine where to place overlay breakpoints.

To support the encoding of additional flags, the matrix is expanded to support additional columns.

- In the program state machine implementation provided by `libdwarf`, these columns are currently individual Dwarf_Small (byte) values.
- The DWARF 3 standard defines the new `prologue_end` and `epilogue_begin` flags.
- Similarly to support IBM z/OS breakpoint type flags (related to program hook opcodes, and the equivalent overlay breakpoints), many further columns are required.

The space required for the expanded `libdwarf` Dwarf_Line array is minimized by changing the attribute flag representation of the expanded matrix to use bit flags. These would be contained in the following new Dwarf_Word flags (4 bytes):

- One with all standard DWARF flags
- One with all platform-specific DWARF flags

To maintain portability, the platform specific attribute flags would be:

- Defined via an enumeration constant whose value represents the bit number (from 0 to 31).
- Encoded in the line number program using a new opcode with a parameter whose value is the enumeration constant for the flag to be set.

The initial state for each row during decoding would be FALSE.

In addition to accommodating the mapping of the current z/OS hook types, it allows for future attribute flag growth.

Symbol declaration coordinates

To define the declaration coordinates for a symbol or type, the standard DWARF provides the attributes `DW_AT_decl_file`, `DW_AT_decl_line`, and `DW_AT_decl_column`. These are referred to by the abbreviation DECL.

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. The absence of the attribute 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 absence of the attribute indicates that no source line has been specified.

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

State machine registers

The line number program state machine is extended with the following registers:

Register	Purpose
relstmtno	An unsigned integer indicating an relative statement on line number where the source statement begins. The value 0 indicates that this field is not used.
system_flag	A Dwarf_Word value indicating the system-dependent attribute flag states.

The numbering of bits within the sysattr_flag value for z/OS is defined by the following DW_SAT_IBM_xxxx enumeration constants:

Attribute	Enumeration	Description
DW_SAT_IBM_hook	0	A hook opcode is present in the generated program.
DW_SAT_IBM_path_label	1	Path label
DW_SAT_IBM_path_call_return	2	Path: call. After return from call
DW_SAT_IBM_alloc	3	Storage allocation
DW_SAT_IBM_autoinit	4	Automatic storage initialization
DW_SAT_IBM_path_do_begin	5	Path: start of do loop
DW_SAT_IBM_path_true_if	6	Path: if statement evaluated TRUE.
DW_SAT_IBM_path_false_if	7	Path: if statement evaluated FALSE.
DW_SAT_IBM_path_when_begin	8	Path: start of case/select/switch statement specific case.
DW_SAT_IBM_path_otherwise	9	Path: start of case/select/switch statement default case.
DW_SAT_IBM_path_postcompound	10	Path: merge of multiple paths.
DW_SAT_IBM_path_call_begin	11	Path: call. After parm list build, before actual call.
DW_SAT_IBM_goto	12	Goto statement
DW_SAT_IBM_block_exit	13	Scope block exit
DW_SAT_IBM_multiexit	14	Scope block multiple exit
DW_SAT_IBM_prologue_begin	15	The location of where the subprogram prolog begins.
DW_SAT_IBM_funcentry	16	The first breakpoint location within a function
DW_SAT_IBM_path_search_when_begin	17	Path: the logic following a WHEN within a COBOL SEARCH is about to be executed.

Attribute	Enumeration	Description
DW_SAT_IBM_path_search_otherwise	18	Path: the logic following an AT END within a COBOL SEARCH is about to be executed.
DW_SAT_IBM_path_declarative_return	19	Path: control is about to return from a declarative procedure (USEAFTER ERROR, etc.)
DW_SAT_IBM_path_not_begin	20	Path: the logic associated with one of the following phrases is about to be executed: • NOT ON SIZE ERROR • NOT ON EXCEPTION • NOT ON OVERFLOW • NOT AT END (other than SEARCH AT END) • NOT AT END-OF-PAGE • NOT INVALID KEY
DW_SAT_IBM_path_not_end	21	Path: the logic following the end of a statement containing one of the following phrases is about to be executed: • NOT ON SIZE ERROR • NOT ON EXCEPTION • NOT ON OVERFLOW • NOT AT END (other than SEARCH AT END) • NOT AT END-OF-PAGE • NOT INVALID KEY
DW_SAT_IBM_synchronization	22	Synchronization point
DW_SAT_IBM_perform_begin	23	A COBOL perform begin block
DW_SAT_IBM_perform_end	24	A COBOL perform end block
DW_SAT_IBM_if	25	IF statement
DW_SAT_IBM_evaluate	26	COBOL EVALUATE statement
DW_SAT_IBM_search	27	COBOL SEARCH statement
DW_SAT_IBM_loop_while	28	`while` loop
DW_SAT_IBM_loop_until	29	`do-while` loop
DW_SAT_IBM_branch_end	30	Last statement for a branch statement

Extended opcodes

The IBM z/OS DWARF extensions define the following additional standard opcode to support platform specific attribute flag extensions:

DW_LNE_IBM_define_global_file
This opcode takes a DIE offset as an operand. It identifies the DW_TAG_IBM_src_file DIE in the global .debug_srcfiles section. This opcode must precede all other opcodes in the line number program except for DW_LNE_define_file.

DW_LNE_IBM_set_system_flag
This opcode takes a single unsigned LEB128 operand and perform a bitwise OR operation with the system flag attribute of the state machine.

DW_LNE_IBM_clear_system_flag
This opcode takes a single unsigned LEB128 operand and perform a bitwise NOT operations and then a bitwise AND operation with the system flag attribute of the state machine.

Dwarf_Line object

The Dwarf_Line object contains an opaque data type that applies to Dwarf_Line data, which can be used as descriptors in searches for source lines.

When it is no longer needed, the storage identified by these descriptors is freed individually, using the dwarf_dealloc operation with the allocation type DW_DLA_LINE. Dwarf_Line data is returned from successful calls to the following operations:

- dwarf_persist_srclines
- dwarf_srclines

Type definition

```
typedef struct Dwarf_Line_s* Dwarf_Line;
```

Consumer operations

The operations in this section are introduced by the program line-number extensions to DWARF.

dwarf_srclines_dealloc operation

The dwarf_srclines_dealloc operation deallocates all memory acquired from dwarf_srclines.

Prototype

```
void
  dwarf_srclines_dealloc(
    Dwarf_Debug           dbg,             /* libdwarf consumer instance    I*/
    Dwarf_Line*           linebuf,         /* List of line number rows      I*/
    Dwarf_Signed          linecount,       /* List entry count              I*/
    Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer instance.

linebuf
Input. This is the list of line number matrix rows obtained from dwarf_srclines()

linecount
Input. This is the number of line number matrix rows obtained from dwarf_srclines().

error
Input/output. This accepts and returns the Dwarf_Error object.

Example

```
Dwarf_Line  *linebuf;
Dwarf_Signed linecount;

/* Get line number table entries */
dwarf_srclines (cudie, &linebuf, &linecount, &err);

/* Add code to process returned line number table entries */

/* Once finished, deallocate memory */
dwarf_srclines_dealloc (dbg, linebuf, linecount);
```

dwarf_pc_linepgm operation

The dwarf_pc_linepgm operation locates the line-number program for a given PC address.

Prototype

```
int dwarf_pc_linepgm (
    Dwarf_Debug               dbg,
    Dwarf_Addr                pc,
    Dwarf_Off*                ret_linepgm_ofs,
   Dwarf_Error*               error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer object.

pc
Input. This accepts a value for the PC.

ret_linepgm_ofs
Output. This returns the line-program offset.

error
Input/output. This accepts and returns the Dwarf_Error object.

Return values

The dwarf_pc_linepgm operation returns DW_DLV_NO_ENTRY if the PC address is not within the range of line-number programs.

dwarf_die_linepgm operation

The dwarf_die_linepgm operation locates the line-number program for a given DIE. The operation navigates towards the root DIE.

dwarf_die_linepgm navigates towards the root DIE. It stops when it locates the CU DIE or partial-unit DIE with the most relevant line-number program.

Prototype

```
int dwarf_die_linepgm(
    Dwarf_Die          die,
    Dwarf_Die*         ret_line_die,
    Dwarf_Off*         ret_linepgm_ofs,
    Dwarf_Error*       error);
```

Parameters

die
Input. This accepts the DIE object.

ret_line_die
Output. This returns the DIE that owns the line-number program.

ret_linepgm_ofs
Output. This returns the offset in .debug_line for the line-number program.

error
Input/output. This accepts and returns the Dwarf_Error object.

Return values

The dwarf_die_linepgm operation returns DW_DLV_NO_ENTRY if the line-number program does not exist.

dwarf_linepgm_offset operation

The dwarf_linepgm_offset operation searches for the line-number program offset attribute (DW_AT_stmt_list) associated with a given DIE.

Prototype

```
int dwarf_linepgm_offset(
  Dwarf_Die             die,
  Dwarf_Off*            returned_offset,
  Dwarf_Error*          error);
```

Parameters

die
Input. This accepts the DIE object.

returned_offset
Output. This returns the .debug_line offset.

error
Input/output. This accepts and returns the Dwarf_Error object.

Return values

The dwarf_linepgm_offset operation returns DW_DLV_NO_ENTRY if the given DIE does not have a DW_AT_stmt_list attribute.

dwarf_line_srcdie operation

The dwarf_line_srcdie operation searches for the source file DIE for a line-matrix row.

Prototype

```
int dwarf_line_srcdie(
    Dwarf_Line          line,
    Dwarf_Die*          ret_die,
    Dwarf_Error*        error);
```

Parameters

line
Input. This accepts the line-number matrix row.

ret_die
Output. This returns the DW_TAG_IBM_srcfile DIE associated with the line-number matrix row.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_line_srcdie operation returns DW_DLV_NO_ENTRY if no line-number information exists.

dwarf_line_isa operation

The dwarf_line_isa operation searches for the instruction set architecture ISA for a line-matrix row.

Prototype

```
int dwarf_line_isa(
    Dwarf_Line                line,
    Dwarf_Unsigned*           ret_isa,
    Dwarf_Error*              error);
```

Parameters

line
Input. This accepts a line number of a matrix row.

ret_isa
Output. This returns the line ISA value.

error
Input/output. This accepts and returns the Dwarf_Error object.

dwarf_line_standard_flags operation

The dwarf_line_standard_flags operation searches for the standard line-attribute flags for a line-matrix row.

Prototype

```
int dwarf_line_standard_flags(
    Dwarf_Line                line,
    Dwarf_Flag*               returned_flags,
    Dwarf_Error*              error);
```

Parameters

line
Input. This accepts a line number of a matrix row.

returned_flags
Output. This returns the standard line flags.

error
Input/output. This accepts and returns the Dwarf_Error object.

dwarf_line_system_flags operation

The dwarf_line_system_flags operation searches for the system specific line attribute-flags for a line matrix row.

Prototype

```
int dwarf_line_system_flags(
    Dwarf_Line               line,
    Dwarf_Flag*              returned_flags,
    Dwarf_Error*             error);
```

Parameters

line
Input. This accepts a line number of a matrix row.

returned_flags
Output. This returns the system line flags.

error
Input/output. This accepts and returns the Dwarf_Error object.

dwarf_linebeginprologue operation

The dwarf_linebeginprologue operation tests if the line-matrix row begins the subprogram prologue.

Prototype

```
int dwarf_linebeginprologue(
    Dwarf_Line                  line,
    Dwarf_Bool*                 returned_bool,
    Dwarf_Error*                error);
```

Parameters

line
Input. This accepts a line number of a matrix row.

returned_bool
Output. This returns the test results.

error
Input/output. This accepts and returns the Dwarf_Error object.

dwarf_lineendprologue operation

The dwarf_lineendprologue operation tests if the line-matrix row ends the subprogram prologue.

Prototype

```
int dwarf_lineendprologue(
    Dwarf_Line                  line,
    Dwarf_Bool*                 returned_bool,
    Dwarf_Error*                error);
```

Parameters

line
Input. This accepts a line number of a matrix row.

returned_bool
Output. This returns the test results.

error
Input/output. This accepts and returns the Dwarf_Error object.

dwarf_lineepilogue operation

The dwarf_lineepilogue operation tests if the line-matrix row begins the subprogram epilogue.

Prototype

```
int dwarf_lineepilogue(
    Dwarf_Line                  line,
```

```
    Dwarf_Bool*                returned_bool,
    Dwarf_Error*               error);
```

Parameters

line
Input. This accepts a line number of a matrix row.

returned_bool
Output. This returns the test results.

error
Input/output. This accepts and returns the `Dwarf_Error` object.

dwarf_persist_srclines operation

The `dwarf_persist_srclines` operation decodes a line-number program into the line-number information matrix. The line-number information matrix is a persistent copy that is associated with the owning compilation unit.

Prototype

```
int dwarf_persist_srclines(
    Dwarf_Die           die,
    Dwarf_Line**        ret_linebuf,
    Dwarf_Signed*       ret_linecount,
    Dwarf_Error*        error);
```

Parameters

die
Input. This accepts the `Dwarf_Die` object with the `DW_AT_stmt_list` attribute.

ret_linebuf
Output. This returns the list of line-number matrix rows.

ret_linecount
Output. This returns the number of line-number matrix rows in the list.

error
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

The `dwarf_persist_srclines` operation returns `DW_DLV_NO_ENTRY` if no line-number information can be found or the DIE does not have the `DW_AT_stmt_list` attribute.

dwarf_pclines operation

The `dwarf_pclines` operation returns one or more line-number entries that match a given PC-line slide argument.

The following list describes what is returned when a given PC-line slide argument is specified:

- If `DW_DLS_NOSLIDE` is specified, then the operation returns a line-number entry with an address that exactly matches the given PC.
- If `DW_DLS_FORWARD` is specified, then the operation returns a line-number entry with an address that is the closest to the given PC, and line-number entries that are greater than and equal to the PC address.
- If `DW_DLS_BACKWARD` is specified, then the operation returns a line-number entry with an address that is the closest to the given PC, and line-number entries that are less than and equal to the PC address.

Prototype

```
int dwarf_pclines(
  Dwarf_Debug           dbg,
  Dwarf_Addr            pc,
  Dwarf_Line**          ret_linebuf,
  Dwarf_Signed          slide,
  Dwarf_Signed*         ret_linecount,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts the libdwarf consumer.

pc
Input. This accepts the PC address.

slide
Input. This accepts the PC-line slide argument.

ret_linebuf
Output. This returns the list of line-number matrix rows.

ret_linecount
Output. This returns the count of the items in the list.

error
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

The `dwarf_pclines` operation returns `DW_DLV_NO_ENTRY` if no line-number entry matches the PC-line slide argument.

Memory allocation

You can deallocate the parameters as required.

Example: The following example is a code fragment that deallocates the `ret_linebuf` parameter:

```
if (dwarf_pclines (dbg, pc, slide, &linebuf, &linecount, &err)
  == DW_DLV_OK)
  dwarf_dealloc (dbg, linebuf, DW_DLA_LIST);
```

Chapter 7. Program source description extension

This section is used by DWARF consumer APIs to identify source files in an application module. It accommodates programs that are built using global optimization compiler options, as well as those compiled as a single compilation unit. Because common source files are recorded in a single object, minimal space is required to represent source files.

Debug section

The `.debug_srcfiles` section contains Debug Information Entries (DIEs), which describe the contents and usage of program source files. This information originates during the program translation process (compile or assembly), and initially describes the source files used for the single CU.

A separate DIE section block is generated for each:

- source file
- include file
- file location information

Each `.debug_srcfiles` section block may share the associated `.debug_abbr` section block, but must have a separate `.rel.debug_srcfiles` relocation section block.

The `.debug_srcfiles` section is a global section and contains DIEs with optional attribute tags. These attribute tags define the globally unique source files for all CUs in the application module. A source file is identified by attributes such as the system name, file name, date and time last modified, type, and file contents (considering macro expansions, conditional compilation, and preprocessor expansion as appropriate). Whenever all attributes are the same, a single entry is used. A difference in one or more of these values results in the creation of a separate entry. If multiple source file DIEs have fields that refer to other DIEs with the same value, the referenced DIE is shared to minimize the size of the DWARF information.

The DWARF file contains the name of each source file that contributed to an object or executable file. Typically, the DWARF file is used by a debugger to locate and open each source file, so that the contents can be retrieved and used to support program source display functions. In the `.debug_info` section, each CU is represented by a DIE with the tag `DW_TAG_compile_unit`. This DIE typically has the following attributes:

- `DW_AT_stmt_list`, with an offset to the CU's line table information in the `.debug_line` section
- `DW_AT_comp_dir`, with the current working directory at the compile time

In the `.debug_line` section, the line data associated with each CU is encoded as a line number program (for more information, refer to *DWARF Debugging Information Format*, V3, Draft 7). The line number program consists of opcodes. These opcodes represent operations in the statement state machine. For more information, refer to *DWARF Debugging Information Format*, V4.

The `DW_LNE_IBM_define_global_file` opcode refers to the source-file entry defined in `.debug_srcfiles` debug section.

Block header

Each block of information in the `.debug_srcfiles` section begins with a header that contains the location-format information. This header does not replace any debugging information entries. It is additional information that is represented outside the standard DWARF tag/attributes format. It is used to navigate the information blocks in the `.debug_srcfiles` section. This is similar in format and intent to the standard Compile-Unit Header for `.debug_info`.

Block length
A 4-byte or 12-byte unsigned integer representing the length of the `.debug_pa` block, not including the length of the field itself. In the 32-bit Dwarf format, this is a 4-byte unsigned integer (which must

be less than 0xFFFFFF00). In the 64-bit format, this is a 12-byte unsigned integer that consists of the 4-byte value 0xFFFFFFFF followed by an 8-byte unsigned integer that gives the actual value of the integer.

DWARF version
A 2-byte unsigned integer representing the version of the DWARF information for that block of `.debug_srcfiles` information.

.debug_abbrev offset
A 4-byte or 8-byte unsigned offset into the `.debug_abbrev` section that associates the `.debug_scrfiles` information with a particular set of debugging information entry abbreviations.

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 .

Section-specific DIEs

The following DIEs could occur within a .debug_srcfiles section:

DW_TAG_IBM_src_location
Identifies the system and primary location of a source file. It is created in a separate .debug_srcfiles block.

DW_TAG_IBM_src_file
Identifies a single globally-unique program source file. It is created in the same .debug_srcfiles block as any child DW_TAG_IBM_src_nest DIEs.

Companion sections

For each block of information in the .debug_srcfiles block, there is an associated block in the debug sections that are listed below

.debug_abbrev
This contains abbreviations-table entries which describe the low-level encoding for each particular form of DIE. The entry is a DIE tag that is optionally associated with a specific grouping of attribute entries. Each attribute has an associated form code which describes the precise encoding of the data for each attribute. For more information, see section 7.5.3 in DWARF Debugging Information Format, V3, Draft 7.

.rel.debug_srcfiles
This contains the ELF-format relocation entries which are used to perform relocations related to the .debug_srcfiles information. These relocation entries are section offsets.

Reference section

DIEs in .debug_info, .debug_line and .debug_srcfiles sections can refer to DIEs in a .debug_srcfiles section.

A source file is described by a DW_TAG_IBM_src_file DIE, which uses a DW_AT_IBM_src_location attribute to specify the location of the source file. This attribute contains the offset within the .debug_srcfiles section of the associated DW_TAG_IBM_src_location DIE. The line number table in .debug_line can use the DW_LNE_IBM_define_global_file opcode to specify the source file that contributes to the line number table. The opcode data value is the .debug_srcfiles section offset of the DW_TAG_IBM_src_file DIE.

Attributes forms

The DWARF attribute form governs how the value of a Debug Information Entry (DIE) attribute is encoded. The IBM extensions to DWARF do not introduce new attribute form codes, but extend their usage.

The Attribute Form Class `srcfileptr` can identify any debugging information entry within a `.debug_srcfiles` section. This type of reference (`DW_FORM_sec_offset` in DWARF V4, `DW_FORM_data4` and `DW_FORM_data8` in DWARF V3) is an offset from the beginning of the `.debug_srcfiles` section.

Source-file entries

Source location entries

The DIE with the tag DW_TAG_IBM_src_location identifies the system and primary location of the source file. The source location DIE is followed by the DW_AT_name attribute. The attribute value is of form DW_FORM_string. This is a null-terminated string that follows the convention used for the standard DWARF DW_LNS_define_file opcode (which means that it consists of the system name, a colon delimiter, and the primary location, which is operating-system-dependent and file-system-dependent.

The following table lists the defined formats for the z/OS environments.

Table 3. Defined formats for the z/OS environments

OS and file system	**Format**
z/OS HFS path name	system:/absolute/hfs/path/name
z/OS MVS™ data set	system://data.set.name
CMS minidisk	system://volume_label
CMS SFS	system://pool:sfs.dir.name
CMS POSIX BFS path name	system:/absolute/bfs/path/name

Source file name entries

The DIE with the DW_TAG_IBM_src_file tag identifies a single globally-unique program source file.

The source-file DIE may be followed by one or more of the following attributes:

- DW_AT_name
- DW_AT_IBM_charset
- DW_AT_IBM_date
- DW_AT_IBM_src_location
- DW_AT_IBM_src_origin
- DW_AT_IBM_src_type

DW_AT_name
This attribute is a string of form DW_FORM_string, and it is a standard DWARF attribute. This optional value is the minor portion of the file name. It is used in combination with the major portion of the file name from the DW_TAG_IBM_src_location DIE at the offset identified by the DW_AT_IBM_src_location attribute. The DW_AT_name attribute is used to complete the location information for the source file. The value is a null-terminated string, in a format which is operating-system and file-system dependent. If the source file is compiler generated, the name can be used to provides a description of the compiler generated file, and not necessarily a physical file name.

Table 4. DW_AT_name formats	
OS and file system	**Format**
z/OS HFS path name	filename.ext
z/OS MVS sequential data set	Attribute is omitted.
z/OS MVS partitioned data set	membername
CMS minidisk	fn.ft.fm
CMS SFS	file.name.ext
CMS POSIX BFS path name	file.name.ext

DW_AT_IBM_charset
This attribute value is a string of form DW_FORM_string. This value indicates the codepage for the program source file. If the attribute is missing on z/OS, the program source file is assumed to be encoded in IBM-1047.

DW_AT_IBM_date
This attribute value is a constant of form DW_FORM_udata. This value represents the date and time of last modification of the file. The base date is the same as that used for the line number program DW_LNE_define_file opcode. This is an optional attribute, because some z/OS files do not have this value available.

DW_AT_IBM_src_location
This attribute value provides the source file location and the attribute encoding is of the class `srcfileptr`. It contains the offset in the `.debug_srcfiles` section for the DW_TAG_IBM_src_location DIE for this file.

DW_AT_IBM_src_origin
This attribute value is a constant of form DW_FORM_data*. The value describes the file system where the program source is located. The following values are defined:

- 0 - Unix file system (including z/OS HFS file system)
- 1 - z/OS sequential data set
- 2 - z/OS partitioned data set
- 3 - z/VM® enhanced disk format (CMS minidisk files)
- 4 - z/VM shared file system
- 5 - z/VM OpenExtensions byte file system
- 6 - z/VSE® file system

DW_AT_IBM_src_type
This attribute value is a constant of form DW_FORM_data*. This value categorizes the program source file into one of the following categories:

- 0 - Primary file
- 1 - User Include file
- 2 - System Include file
- 3 - Compiler generated file

DW_AT_artificial
Any source file that does not participate in the line number table, (i.e. only used for declaration of object or type) may have this attribute, which is a flag.

DW_AT_IBM_md5
Contains a 16 byte MD5 signature that uniquely identifies the source file. The form of the attribute is DW_FORM_block1 containing a 16 byte value.

DW_AT_IBM_src_text
This attribute value provides the source text content and the attribute encoding is of the class `srctextptr`. It contains the offset in the .debug_srctext section for this file. (refer to "Source text extension" for more info)

DW_AT_IBM_src_attr
This attribute value provides the source attribute and the attribute encoding is of the class `srcattrptr`. It contains the offset in the `.debug_srcattr` section for this file. For more information, see Chapter 9, “Program source attribute extensions,” on page 109.

Callback functions

Dwarf_Retrieve_Srcline_CBFunc object

This object contains a prototype for a callback function that returns the source line. The user-supplied function is called when the debugging information does not include captured source file information. The callback function must be defined before the `dwarf_get_srcline_given_filename` operation is called.

Type definition

```
typedef int (*Dwarf_Retrieve_Srcline_CBFunc) (
   char*                  filename,
   Dwarf_Unsigned         lineno,
   Dwarf_IBM_charset_type charset,
   char**                 r_srcline,
   int*                   errorcode);
```

Parameters

filename
Input. This accepts the path and filename (/pathname/filename).

lineno
Input. This accepts the required line number.

charset
Input. This accepts the type of the source-file character set.

r_srcline
Output. This returns the source line data.

errorcode
Output. This returns the error code.

Dwarf_Retrieve_Srcline_term_CBFunc object

This object contains a prototype for a callback function that frees the storage allocated for the data source line returned by the Dwarf_Retrieve_Srcline_CBFunc callback function. The callback function must be defined before the `dwarf_get_srcline_given_filename` operation is called.

Type definition

```
typedef void (*Dwarf_Retrieve_Srcline_term_CBFunc)(
    char*     srcline);
```

Parameters

srcline
Input. This accepts the source line returned by the Dwarf_Retrieve_Srcline_CBFunc function.

Dwarf_Retrieve_Srccount_CBFunc object

This object contains the prototype for a callback function that returns the count of source lines. The function is called when the debugging informatio does not contain captured source. The callback function must be defined before the `dwarf_get_srcline_given_filename` operation is called.

Type definition

```
typedef int (*Dwarf_Retrieve_Srccount_CBFunc) (
   char*                  filename,
   Dwarf_IBM_charset_type  charset,
   Dwarf_Unsigned*         r_srccnt,
   int*                    errorcode);
```

Parameters

filename
Input. This accepts the path and filename (/pathname/filename).

charset
Input. This accepts the type of the source-file character set.

r_srccnt
Output. This returns the number of source lines.

errorcode
Output. This returns the error code.

Source-file consumer operations

This section describes the operations that are used to access debug information using information found within .debug_srcfiles

dwarf_get_srcdie_given_filename operation

The `dwarf_get_srcdie_given_filename` operation searches all `DW_TAG_IBM_src_file` DIEs for a `DW_AT_name` field that matches the given filename.

Prototype

```
int dwarf_get_srcdie_given_filename (
  Dwarf_Debug            dbg,
  const char*            filename,
  Dwarf_Die**            ret_sfdies,
  Dwarf_Unsigned*        ret_diecount,
  Dwarf_Error*           error);
```

Parameters

dbg
Input. This accepts a `libdwarf` consumer object.

`filename`
Input. This accepts a short filename, without a path. The format is *`filename`*.

`ret_sfdies`
Output. This returns the source file DIEs that match the `filename`.

`ret_diecount`
Output. This returns the count of the `ret_sfdies`.

`error`
Input/output. This accepts and returns the `Dwarf_Error` object.

Return values

The dwarf_get_srcdie_given_filename operation returns DW_DLV_NO_ENTRY if none of the DW_TAG_IBM_src_file DIEs matches the given filename.

Memory allocation

The list object ret_sfdies and its elements are persistent copies that are associated with the owning libdwarf consumer object, and must be deallocated only by dwarf_finish().

dwarf_srclines_given_srcdie operation

The dwarf_srclines_given_srcdie operation identifies all the Dwarf_Line objects that are associated with the given Dwarf_Die object.

The Dwarf_Die object must be a DW_TAG_IBM_src_file DIE. The returned Dwarf_Line objects are sorted in ascending order first by line number, then by PC address.

Prototype

```
int dwarf_srclines_given_srcdie (
  Dwarf_Debug           dbg,
  Dwarf_Die             sf_die,
  Dwarf_Line**          ret_linebuf,
  Dwarf_Signed*         ret_linecount,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer object.

sf_die
Input. This accepts the DW_TAG_IBM_src_file DIE.

ret_linebuf
Output. This returns a list of the line-number matrix rows in the given sf_die.

ret_linecount
Output. This returns the count of the rows in sf_die.

error
Input/output. This accepts and returns the Dwarf_Error object.

Return values

The dwarf_srclines_given_srcdie operation returns DW_DLV_NO_ENTRY if there are no Dwarf_Line objects that reference the given sf_die.

Memory allocation

The list object ret_linebuf and its elements are persistent copies that are associated with the owning libdwarf consumer object, and must be deallocated only by dwarf_finish().

dwarf_get_srcline_given_filename operation

The dwarf_get_srcline_given_filename operation searches a given file and returns the content of the specified source line.

Prototype

```
int dwarf_get_srcline_given_filename(
  Dwarf_Debug               dbg,
```

```
  char*                     longfn,
  Dwarf_IBM_charset_type    charset,
  Dwarf_Unsigned            lineno,
  char**                    ret_srcline,
  Dwarf_Error*              error);
```

Parameters

dbg
Input. This accepts a `libdwarf` consumer object.

longfn
Input. This accepts a path and filename. The format is *system:/pathname/filename*.

charset
Input. This accepts the character-set type of the `longfn` file.

lineno
Input. This accepts the line number of the required source line. Note that the line numbering starts from 1 and not 0.

ret_srcline
Output. This returns the source line.

error
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

The `dwarf_get_srcline_given_filename` operation returns `DW_DLV_NO_ENTRY` if it cannot find the file or the line number does not exist.

Memory allocation

You can deallocate the parameters as required.

Example: A code fragment that deallocates the `ret_srcline` parameter:

```
if (dwarf_get_srcline_given_filename (dbg, ..., &ret_srcline, &err)
  == DW_DLV_OK) {
  dwarf_dealloc (dbg, ret_srcline, DW_DLA_STRING);
}
```

Note: For reasons of clarity, not all the parameters have been entered in the above code. Unlisted parameters are represented by ellipses (...).

For more information about deallocating the `error` parameter, see *Consumer Library Interface to DWARF*, by the UNIX International Programming Languages Special Interest Group.

dwarf_get_srcline_count_given_filename operation

The `dwarf_get_srcline_count_given_filename` operation counts the lines within a source file.

Prototype

```
int dwarf_get_srcline_count_given_filename(
  Dwarf_Debug             dbg,
  char*                   longfn,
  Dwarf_IBM_charset_type  charset,
  Dwarf_Unsigned*         ret_linecount,
  Dwarf_Error*            error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer object.

longfn
Input. This accepts a long filename. The format is *system:/pathname/filename*.

charset
Input. This accepts the character-set type of the longfn file.

ret_linecount
Output. This returns the total number of lines within a specified source file.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_get_srcline_count_given_filename operation returns DW_DLV_NO_ENTRY if the file is empty.

dwarf_register_src_retrieval_callback_func operation

The dwarf_register_src_retrieval_callback_func operation registers the user-defined source-retrieval functions.

The dwarf_register_src_retrieval_callback_func operation is called when captured source is not available within the debugging information.

This operation refers to callback functions that are based on the following prototypes:

- Dwarf_Retrieve_Srcline_CBFunc
- Dwarf_Retrieve_Srcline_term_CBFunc
- Dwarf_Retrieve_Srccount_CBFunc

Prototype

```
int dwarf_register_src_retrieval_callback_func(
  Dwarf_Debug                          dbg,
  Dwarf_Retrieve_Srcline_CBFunc        rs_f,
  Dwarf_Retrieve_Srcline_term_CBFunc   termrs_f,
  Dwarf_Retrieve_Srccount_CBFunc       rsc_f,
  Dwarf_Error*                         error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer object.

rs_f
Input. This accepts the name of a function that is of the Dwarf_Retrieve_Srcline_CBFunc type.

termrs_f
Input. This accepts the name of a function that is of the Dwarf_Retrieve_Srcline_term_CBFunc type.

rsc_f
Input. This accepts the name of a function that is of the Dwarf_Retrieve_Srccount_CBFunc type.

error
Input/output. This accepts and returns the Dwarf_Error object.

Chapter 8. Program source text extensions

This section is used to hold the contents of the source files or compiler generated source. A source-level debugger might need to display the user source when the original source file is not available on the system.

Debug section

The `.debug_srctext` section contains the source text. A separate block is generated for each primary source file and each include file. Each block contains a block header followed by the source text, which is encoded in UTF-8 and compressed with `zlib`. The end of each line is delimited with the UTF-8 character `'\n'` (codepoint: 0x0A).

Block header

Each block of information in the .debug_srctext section begins with a header, which consists of the following information:

Block length
This holds the total length of the compressed source text, and the section header, not including the block length field itself. It is also used to determine whether this block of information is 32-bit DWARF format or 64-bit DWARF format. In the 32-bit DWARF format, the first 4-byte is an unsigned integer representing the block length (which must be less than 0xFFFFFF00). In the 64-bit DWARF format, the first 4-byte is 0xFFFFFFFF, and the following 8 bytes is an unsigned integer representing the block length.

In the 64-bit DWARF format, this is a 12-byte unsigned integer, and it has two parts:

- The first 4 bytes have the value 0xFFFFFFFF.
- The following 8 bytes contain the actual length represented as an unsigned 64-bit integer.

Version field
A 2-byte unsigned integer represents the version of the .debug_srctext information for the block. This version is specific to the .debug_srctext section. The currently supported version is 0x0001.

Header length
The number of bytes following the header_length field to the beginning of the first byte of the compressed source text. In the 32-bit DWARF format, it is a 4-byte unsigned length; in the 64-bit DWARF format, this field is an 8-byte unsigned length.

Eye catcher
A 2-byte eye catcher to help identify the boundaries of different source text sections. The value should be 0xCDA6.

Data size
An 8-byte unsigned integer representing the size of the original source text after it has been uncompressed.

Reference section

DIEs in the .debug_srcfiles section can refer to the source text in the .debug_srctext section.

A source file is described by a `DW_TAG_IBM_src_file` DIE, which can have a `DW_AT_IBM_src_text` attribute that points to the start of the source text stream in the .debug_srctext section.

Attributes forms

The DWARF attribute form governs how the value of a Debug Information Entry (DIE) attribute is encoded. The IBM extensions to DWARF do not introduce new attribute form codes, but extend their usage.

The Attribute Form Class `srctextptr` can identify any source text block within a `.debug_srctext` section. This type of reference (`DW_FORM_sec_offset` in DWARF V4, `DW_FORM_data4` and `DW_FORM_data8` in DWARF V3) is an offset from the beginning of the `.debug_srctext` section.

Source text consumer operations

The operations in this section retrieve and manipulate information within the `.debug_srctext` section.

dwarf_access_source_text operation

The `dwarf_access_source_text` operation retrieves the source data embedded in .debug_srctext. The returned source text information is encoded in the codeset specified by dwarf_set_codeset. Source lines are delimited by the `'\n'` character.

Prototype

```
int dwarf_access_source_text(
    Dwarf_Die           die,
    Dwarf_Unsigned*     ret_numlines,
    Dwarf_Off**         ret_lineoff,
    char**              ret_srclines,
    Dwarf_Unsigned*     ret_srclen,
    Dwarf_Error*        error);
```

Parameters

`die`
Input. This accepts the `Dwarf_Die` object that contains the `DW_AT_IBM_src_text` attribute.

`ret_numlines`
Output. This returns the number of lines stored within the source text.

`ret_lineoff`
Output. This returns an array that contains byte offsets, relative to the start of *ret_srclines, to the start of each source line.

`ret_srclines`
Output. This returns a contiguous block of memory that contains the entire source text referenced by DIE. It is encoded in the codeset specified by dwarf_set_codeset.

`ret_srclen`
Output. This returns the length of *ret_srclines.

`error`
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

The `dwarf_access_source_text` operation returns `DW_DLV_NO_ENTRY` if the DIE does not have the `DW_AT_IBM_src_text` attribute or the .debug_srctext section is not available.

Source text producer operations

The operations in this section create content in the .debug_srctext section.

dwarf_add_source_text operation

The dwarf_add_source_text operation embeds source data in .debug_srctext, and adds a DW_AT_IBM_src_text attribute in the given DIE. The value of the attribute contains the offset in .debug_srctext that contains the embedded source data.

Prototype

```
int dwarf_add_source_text (
    Dwarf_P_Debug       dbg,
    Dwarf_P_Die         die,
    char*               buf,
    Dwarf_Unsigned      buflen,
    Dwarf_Error*        error);
```

Parameters

dbg
Input. This accepts the Dwarf_P_Debug object.

die
Input. This accepts the Dwarf_Die object that contains the DW_AT_IBM_src_text attribute.

buf
Input. This accepts the source data buffer encoded in UTF-8.

buflen
Input. This accepts the length of source data buffer.

error
Input/output. This accepts and returns the Dwarf_Error object.

Return values

The dwarf_add_source_text operation never returns DW_DLV_NO_ENTRY.

Chapter 9. Program source attribute extensions

This section contains source fragment information about a source file. A source line can contain multiple source fragments. A source fragment, such as an executable statement, a compiler directive, or other information such as a comment, can have one or more attributes.

A source-level debugger might need to know a list of variables or expressions that are referenced on a given statement. Knowing this would enable the debugger user to automatically monitor the list of variables or expressions that are referenced on the currently running statement.

A source-level debugger might need to know if a given source fragment is a compiler directive. In that case, it could place emphasis on the source fragment when displaying the source view to the user.

Debug section

The `.debug_srcattr` section contains a table of source fragment entries, which describe the source attributes for one program source file.

If space were not a consideration, the information provided in the `.debug_srcattr` section could be represented as a large matrix, with one row for each source fragment. The matrix would have columns for:

- the source line number
- the source column number
- the source type (for example, comment and executable statement.)
- the offset to `DW_TAG_IBM_xreflist` DIE representing a list of variables or expressions
- and so on

The matrix would also be sorted according to source line number and then source column number to allow for efficient searching. Such a matrix, however, would be impractically large. A byte-codeded language for a state machine is designed to shrink the matrix and store a stream of bytes in the object file instead of the matrix (similar to `.debug_line`). This language can be much more compact than the matrix. When a consumer of the source meta information executes, it must "run" the state machine to generate the matrix for each source file it is interested in. The concept of an encoded matrix also leaves room for expansion. In the future, columns can be added to the matrix to encode other things that are related to individual source statements.

Definitions

The following terms are used in the description of the source attribute information format:

state machine
The hypothetical machine used by a consumer of the source meta information to expand the byte-coded instruction stream into a matrix of source meta information.

source attribute program
A series of byte-coded source meta information representing one source file.

State machine registers

The source attribute information state machine has the following registers:

line
An unsigned integer indicating a source line number where the source statement begins. Lines are numbered beginning at 1.

column
A signed integer indicating a column number where the source statement begins. Columns are numbered beginning at 1. The value -1 indicates that this field is not used.

xreflist
A DIE index into `.debug_xref`. This locates the DW_TAG_IBM_xreflist DIE which contains a list of variables or expressions being referenced by this source fragment. All DW_TAG_IBM_xreflist DIEs within the same source attribute program must appear within the same unit section. The value 0 indicates that this field is not used. (The first DIE within a `.debug_xref` section is never a DW_TAG_IBM_xreflist DIE.)

type
Source type (for example, comments or compiler directive). The value 0 indicates that this field is not used.

altline
An unsigned integer indicating an alternate user-specified line number where the source statement begins. The value 0 indicates that this field is not used.

relstmtno
An unsigned integer indicating an relative statement on line number where the source statement begins. The value 0 indicates that this field is not used.

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

- line: 1
- column: -1
- xreflist: 0
- type: 0
- altline: 0
- relstmtno: 0

Source attribute program instructions

The state machine instructions in a source attribute program belong to one of following categories:

special opcodes
These instructions have a ubyte opcode field and no operands. The purpose is to provide a compact way to advance line and xreflist information.

standard opcodes
These instructions have a ubyte opcode field which may be followed by zero or more LEB128 operands. The opcode implies the number of operands and their meanings, but the source attribute program header also specifies the number of operands for each standard opcode.

extended opcodes
These instructions 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).

Source attribute program header

The optimal encoding of source attribute information depends to a certain degree upon the structure of the source program. The source attribute program header provides information used by consumers in decoding the source attribute program instructions for a particular source file and also provides information used throughout the rest of the source attribute program.

The source attribute program for each source file begins with a header containing the following fields in order:

unit_length
A 4-byte or 12-byte unsigned integer represents the size in bytes of the source attribute information for this source file. This does not include the length of the field itself. In the 32-bit DWARF format this is a 4-byte unsigned integer (which must be less than 0xFFFFFF00). In the 64-bit DWARF format, this is a 12 byte unsigned integer, and it has two parts:

- The first 4 bytes have the value 0xFFFFFFFF.
- The following 8 bytes contains the actual length represented as an unsigned 64-bit integer.

version
A 2-byte unsigned integer represents the version number. This number is specific to the source attribute information and is independent of the DWARF version number. Currently it is 2.

header_length
An unsigned integer represents the number of bytes following this field to the beginning of the first byte of the source attribute information itself. 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.

eyecatcher
A 2-byte eye-catcher. Expected value: 0xCDA7.

When version is 2, the block header also contains the following fields in order:

debug_xref_offset (section offset)
A 4-byte or 8-byte offset into the `.debug_xref` 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. This unit header offset contains all the DW_TAG_IBM_xreflist DIEs that are referenced for this source attribute program.

dieidx_base (sbyte)
This parameter affects the meaning of the special opcodes. See below.

dieidx_range (ubyte)
This parameter affects the meaning of the special opcodes. See below.

opcode_base (ubyte)
The number assigned to the first special opcode. If opcode_base is less than the highest-numbered standard opcode, then standard opcode numbers greater than or equal to the opcode_base are not used in the source attribute program of this unit (and the codes are treated as special opcodes). If opcode_base is greater than the highest-numbered standard opcode, the numbers between that of the highest-numbered standard opcode and the first special opcode (not inclusive) are used for vendor specific extensions.

standard_opcode_length (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.

Source attribute program

As stated before, the goal of a source attribute program is to build a matrix representing one source file. The `line` number may only increase.

Special Opcodes

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

- Add an unsigned integer to the `line` register.
- Add a signed integer to the `xreflist` register.
- Append a row to the matrix using the current values of the state machine registers.

All of the special opcodes do the above things. They differ from one another only in what values they add to the `line` and `xreflist` registers.

Instead of assigning a fixed meaning to each special opcode, the source attribute program uses several parameters in the header to configure the instruction set. There are two reasons for this. First, the opcode space available for special opcodes now ranges from 6 through 255, but the lower bound might increase if one adds new standard opcodes. Thus, the opcode_base field of the source attribute program header gives the value of the first special opcode. Second, the best choice of special-opcode meaning depends on the source file. For example, a source file may have source fragments, each referencing at least 5 symbols. It is advantageous to trade away the ability to increase the `line` register in return for the ability to add larger positive values to the `xreflist` register. For compilers that do not use the `xreflist` register, it is advantageous to trade away the ability to increase `xreflist` register for the ability to add larger positive values to the `line` register. To permit this variety of strategies, the source attribute program header defines a `dieidx_range` field that defines the range of values it can add to the `xreflist` register.

A special opcode value is chosen based on the amount that needs to be added to the `line` and `xreflist` registers. The maximum `xreflist` increment for a special opcode is (`dieidx_base` + `dieidx_range` - 1). If the desired `xreflist` increment is greater than the maximum `xreflist` increment, a standard opcode must be used instead of a special opcode. The special opcode is then calculated using the following formula:

```
opcode = (desired xreflist increment - dieidx_base) +
(dieidx_range * desired line increment) +
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 new `line` and `xreflist` values are given by the following formula:

```
adjusted opcode = opcode - opcode_base
line increment = adjusted opcode / dieidx_range
xreflist increment = dieidx_base + (adjusted opcode % dieidx_range)
```

As an example, suppose that the `opcode_base` is 13, `dieidx_base` is 1, `dieidx_range` is 12. This means that a special opcode can be used whenever two successive rows in the matrix have `xreflist` DIE indexes differing by any value within the range [1, 12] and (because of the limited number of opcodes available) when the difference between source line number is within the range [0, 20], but not all line advances are available for the maximum `xreflist` advance. The opcode mapping would be:

```
        \               xreflist Increment
Line     \
Increment \  1   2   3   4   5   6   7   8   9  10  11  12
---------  ------------------------------------------------
        0   13  14  15  16  17  18  19  20  21  22  23  24
        1   25  26  27  28  29  30  31  32  33  34  35  36
        2   37  38  39  40  41  42  43  44  45  46  47  48
        3   49  50  51  52  53  54  55  56  57  58  59  60
        4   61  62  63  64  65  66  67  68  69  70  71  72
        5   73  74  75  76  77  78  79  80  81  82  83  84
        6   85  86  87  88  89  90  91  92  93  94  95  96
        7   97  98  99 100 101 102 103 104 105 106 107 108
        8  109 110 111 112 113 114 115 116 117 118 119 120
        9  121 122 123 124 125 126 127 128 129 130 131 132
       10  133 134 135 136 137 138 139 140 141 142 143 144
       11  145 146 147 148 149 150 151 152 153 154 155 156
       12  157 158 159 160 161 162 163 164 165 166 167 168
       13  169 170 171 172 173 174 175 176 177 178 179 180
       14  181 182 183 184 185 186 187 188 189 190 191 192
       15  193 194 195 196 197 198 199 200 201 202 203 204
       16  205 206 207 208 209 210 211 212 213 214 215 216
       17  217 218 219 220 221 222 223 224 225 226 227 228
       18  229 230 231 232 233 234 235 236 237 238 239 240
       19  241 242 243 244 245 246 247 248 249 250 251 252
       20  253 254 255
```

Standard Opcodes

The standard opcodes, their applicable operands, and the actions performed by these opcodes are as follows:

DW_SAS_copy
The DW_SAS_copy opcode takes no operands. It appends a row to the matrix using the current values of the state machine registers.

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

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

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

DW_SAS_set_type_flag
The DW_SAS_set_type_flag opcode takes a single unsigned LEB128 operand and perform a bitwise OR operation with the `type` register of the state machine.

DW_SAS_clear_type_flag
The DW_SAS_set_type_flag opcode takes a single unsigned LEB128 operand and perform a bitwise NOT operations and then a bitwise AND operation with the `type` register of the state machine.

DW_SAS_advance_altline
The DW_SAS_advance_altline opcode takes a single unsigned LEB128 operand and adds that value to the `altline` register of the state machine.

Extended Opcodes

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

Attributes forms

The DWARF attribute form governs how the value of a Debug Information Entry (DIE) attribute is encoded. The IBM extensions to DWARF do not introduce new attribute form codes, but extend their usage.

The Attribute Form Class `srcattrptr` can identify any source text block within a `.debug_srcattr` section. This type of reference (`DW_FORM_sec_offset` in DWARF V4, `DW_FORM_data4` and `DW_FORM_data8` in DWARF V3) is an offset from the beginning of the `.debug_srcattr` section.

Consumer operations

The operations in this section retrieve and manipulate information within the `.debug_srcattr` debug section.

dwarf_srcattr_get_version operation

The `dwarf_srcattr_get_version` operation returns the version number for the content within `.debug_srcattr`.

Prototype

```
int dwarf_srcattr_get_version(
    Dwarf_Die           die,
```

```
    Dwarf_Half*         ret_version,
    Dwarf_Error*        error);
```

Parameters

die
Input. DIE containing the DW_AT_IBM_src_attr attribute.

ret_version
Output. Version number of the content that is referenced by the DW_AT_IBM_src_attr attribute.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

DW_DLV_OK
The version number of the .debug_srcattr content referenced by DW_AT_IBM_src_attr is found successfully.

DW_DLV_NO_ENTRY
DIE does not have the DW_AT_IBM_src_attr attribute.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- The given die is NULL.
- The given die does not contain CU context information.
- The given die is corrupted. Cannot determine which debug section the die belongs to.
- Cannot locate a DWARF debug instance associated with the given die.
- Cannot locate the .debug_srcattr section, or the .debug_srcattr section is empty.
- The given ret_version is NULL.
- The length of the encoded text in .debug_srcattr is too large.

dwarf_srcattr_get_altline_used operation

The dwarf_srcattr_get_altline_used operation returns whether the altline register in the .debug_srcattr section is used.

Prototype

```
int dwarf_srcattr_get_altline_used(
    Dwarf_Die           die,
    Dwarf_Bool*         ret_altline_used,
    Dwarf_Error*        error);
```

Parameters

die
Input. DIE containing the DW_AT_IBM_src_attr attribute.

ret_altline_used
Output. Returns whether the altline register is used.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

DW_DLV_OK
The boolean value is returned indicating whether the alternate line number register is used in the `.debug_srcattr` section.

DW_DLV_NO_ENTRY
DIE does not have the `DW_AT_IBM_src_attr` attribute.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- The given `die` is NULL.
- The given `die` does not contain CU context information.
- The given `die` is corrupted. Cannot determine which debug section the `die` belongs to.
- Cannot locate a DWARF debug instance associated with the given `die`.
- Cannot locate the `.debug_srcattr` section, or the `.debug_srcattr` section is empty.
- The given `ret_altline_used` is NULL.
- The length of the encoded text in `.debug_srcattr` is too large.

dwarf_srcattr_get_altlines operation

The `dwarf_srcattr_get_altlines` operation returns an array of alternate line number entries. The array is index by source line number (index 0 corresponds to source line number 1). Each array entry contains the alternate line number. If not available, the entry contains the value 0.

Prototype

```
int dwarf_srcattr_get_altlines(
    Dwarf_Die           die,
    Dwarf_Unsigned**    ret_altlines,
    Dwarf_Unsigned*     ret_numlines,
    Dwarf_Error*        error);
```

Parameters

die
Input. DIE containing the `DW_AT_IBM_src_attr` attribute.

ret_altlines
Output. Array of alternate line number entries indexed by source line number. Index 0 corresponds to source line number 1.

ret_numlines
Output. Number of entries within the returned array.

error
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

DW_DLV_OK
The returned array contains the alternate line number for each source line.

DW_DLV_NO_ENTRY

- DIE does not have the `DW_AT_IBM_src_attr` attribute.
- The `altline` register is not used in the `.debug_srcattr` section.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- The given `die` is NULL.

- The given die does not contain CU context information.
- The given die is corrupted. Cannot determine which debug section the die belongs to.
- Cannot locate a DWARF debug instance associated with the given die.
- Cannot locate the .debug_srcattr section, or the .debug_srcattr section is empty.
- The given ret_altlines or ret_numlines is NULL.
- The length of the encoded text in .debug_srcattr is too large.

dwarf_srcattr_map_altline_to_line operation

The dwarf_srcattr_map_altline_to_line operation maps an alternate line number to a source line number.

Prototype

```
int dwarf_srcattr_map_altline_to_line(
    Dwarf_Die           die,
    Dwarf_Unsigned      altline,
    Dwarf_Unsigned*     ret_lineno,
    Dwarf_Error*        error);
```

Parameters

die
Input. DIE containing the DW_AT_IBM_src_attr attribute.

altline
Iutput. Alternate line number.

ret_lineno
Output. Source line number corresponding to the given alternate line number.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

DW_DLV_OK
The source line number corresponding to the given alternate line number is returned.

DW_DLV_NO_ENTRY

- DIE does not have the DW_AT_IBM_src_attr attribute.
- The altline register is not used in the .debug_srcattr section.
- The specified alternate line number does not exist in the .debug_srcattr section.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- The given die is NULL.
- The given die does not contain CU context information.
- The given die is corrupted. Cannot determine which debug section the die belongs to.
- Cannot locate a DWARF debug instance associated with the given die.
- Cannot locate the .debug_srcattr section, or the .debug_srcattr section is empty.
- The given altline or ret_lineno is NULL.
- The length of the encoded text in .debug_srcattr is too large.

dwarf_srcfrags_given_srcdie operation

The dwarf_srcfrags_given_srcdie operation runs the state machine referenced in the DW_AT_IBM_src_attr attribute of the given DIE. It stores each row of the source attribute program matrix into its own source fragment object (Dwarf_SrcFrag). The returned source fragment objects are ordered as they are ordered in the source attribute program matrix.

Prototype

```
int dwarf_srcfrags_given_srcdie (
    Dwarf_Die           sf_die,
    Dwarf_SrcFrag**     ret_sfragbuf,
    Dwarf_Unsigned*     ret_sfragcount,
    Dwarf_Error*        error);
```

Parameters

sf_die
Input. DIE containing the DW_AT_IBM_src_attr attribute.

ret_sfragbuf
Output. Returned array of source fragment objects.

ret_sfragcount
Output. Number of entries in the returned array of source fragment objects.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

DW_DLV_OK
All source fragment objects associated with the given sf_die is returned.

DW_DLV_NO_ENTRY
No source fragment objects are found within the .debug_srcattr section.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- The given die is NULL.
- The given die does not contain CU context information.
- The given die is corrupted. Cannot determine which debug section the die belongs to.
- Cannot locate a DWARF debug instance associated with the given die.
- Cannot locate the .debug_srcattr section, or the .debug_srcattr section is empty.
- The given ret_sfragbuf or ret_sfragcount is NULL.
- The length of the encoded text in .debug_srcattr is too large.
- An error is encountered when decoding the source attribute program state machine.
- The .debug_srcattr version is not supported.
- Cannot allocate memory to store the decoded source attribute information.

Cleanup

The source fragment object returned by this API is a persistent copy and is associated with the owning compilation unit. It can only be deallocated using one of the following calls:

- dwarf_srcfrag_xref_dealloc()
- dwarf_finish()

dwarf_srcfrags_stmtcount_given_line operation

The dwarf_srcfrags_stmtcount_given_line operation searches through the source fragment objects stored in the source attribute program matrix, and returns the number of executable source statements in the given line number. The line number is 1-based.

Prototype

```
int dwarf_srcfrags_stmtcount_given_line (
    Dwarf_Die           sf_die,
    Dwarf_Unsigned      line_no,
    Dwarf_Unsigned*     ret_stmt_count,
    Dwarf_Error*        error);
```

Parameters

sf_die
Input. DIE containing the DW_AT_IBM_src_attr attribute.

line_no
Input. Source line number.

ret_stmt_count
Output. Number of source fragment objects marked with DW_IST_executable on the given line.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

DW_DLV_OK
Number of source fragment objects with DW_IST_executable is returned.

DW_DLV_NO_ENTRY

- No source fragment object is found within the .debug_srcattr section.
- No source fragment object is associated with the given line number.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- The given die is NULL.
- The given die does not contain CU context information.
- The given die is corrupted. Cannot determine which debug section the die belongs to.
- Cannot locate a DWARF debug instance associated with the given die.
- Cannot locate the .debug_srcattr section, or the .debug_srcattr section is empty.
- The given ret_stmt_count is NULL.
- The length of the encoded text in .debug_srcattr is too large.
- An error is encountered when decoding the source attribute program state machine.
- The .debug_srcattr version is not supported.
- Cannot allocate memory to store the decoded source attribute information.

dwarf_srcfrag_given_line_stmt operation

Given a line number and executable statement count, the dwarf_srcfrag_given_line_stmt operation searches through the source fragment objects stored in the source attribute program matrix, and returns the source fragment object that matches the search criteria. Both the line number and statement number are 1-based. Statement number restarts from 1 at the beginning of each line and increments by one for every executable statement encountered on the same source line.

Prototype

```
int dwarf_srcfrag_given_line_stmt (
    Dwarf_Die           sf_die,
    Dwarf_Unsigned      line_no,
    Dwarf_Unsigned      stmt_no,
    Dwarf_SrcFrag*      ret_sfrag,
    Dwarf_Error*        error);
```

Parameters

sf_die
Input. DIE containing the DW_AT_IBM_src_attr attribute.

line_no
Input. Source line number.

stmt_no
Input. Executable statement number, which restarts from 1 at the beginning of each line and increments by one for every executable statement encountered on the same source line.

ret_sfrag
Output. Source fragment objects marked with DW_IST_executable on the given line and statement number.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

DW_DLV_OK
The source fragment object with DW_IST_executable matching the given line number and statement number is returned.

DW_DLV_NO_ENTRY

- No source fragment object is found within the .debug_srcattr section.
- No source fragment object is associated with the given line number and statement number.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- The given die is NULL.
- The given die does not contain CU context information.
- The given die is corrupted. Cannot determine which debug section the die belongs to.
- Cannot locate a DWARF debug instance associated with the given die.
- Cannot locate the .debug_srcattr section, or the .debug_srcattr section is empty.
- The given ret_sfragbuf or ret_sfragcount is NULL.
- No current source attribute table is defined.
- The length of the encoded text in .debug_srcattr is too large.
- An error is encountered when decoding the source attribute program state machine.
- The .debug_srcattr version is not supported.
- Cannot allocate memory to store the decoded source attribute information.

Cleanup

The source fragment object returned by this API is a persistent copy and is associated with the owning compilation unit. It can only be deallocated using one of the following calls:

- dwarf_srcfrag_xref_dealloc()

- dwarf_finish()

dwarf_srcfrag_line operation

The dwarf_srcfrag_line operation retrieves the line number associated with the source fragment object.

Prototype

```
int dwarf_srcfrag_line(
    Dwarf_SrcFrag       srcfrag,
    Dwarf_Unsigned*     ret_line,
    Dwarf_Error*        error);
```

Parameters

srcfrag
Input. Input source fragment object.

ret_line
Output. Line number associated with the given source fragment object.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

DW_DLV_OK
The line number associated with the source fragment object is returned.

DW_DLV_NO_ENTRY
Never returned.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- The given srcfrag is NULL.
- The given ret_line is NULL.

dwarf_srcfrag_column operation

The dwarf_srcfrag_column operation retrieves the column number associated with the source fragment object. If the column information is unavailable, column value of -1 is returned.

Prototype

```
int dwarf_srcfrag_column(
    Dwarf_SrcFrag       srcfrag,
    Dwarf_Signed*       ret_column,
    Dwarf_Error*        error);
```

Parameters

srcfrag
Input. Input source fragment object.

ret_column
Output. Column number associated with the given source fragment object.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

DW_DLV_OK
The column number associated with the source fragment object is returned.

DW_DLV_NO_ENTRY
Never returned.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- The given `srcfrag` is NULL.
- The given `ret_column` is NULL.

dwarf_srcfrag_altline operation

The `dwarf_srcfrag_altline` operation retrieves the alternative line number associated with the source fragment object.

Prototype

```
int dwarf_srcfrag_altline(
    Dwarf_SrcFrag       srcfrag,
    Dwarf_Unsigned*     ret_altline,
    Dwarf_Error*        error);
```

Parameters

srcfrag
Input. Input source fragment object.

ret_altline
Output. Alternative line number associated with the given source fragment object.

error
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

DW_DLV_OK
The alternative line number associated with the source fragment object is returned.

DW_DLV_NO_ENTRY
Never returned.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- The given `srcfrag` is NULL.
- The given `ret_altline` is NULL.

dwarf_srcfrag_typeflag operation

The `dwarf_srcfrag_typeflag` operation retrieves the type flag associated with the source fragment object. The supported type flags are listed in `Dwarf_IBM_srcattr_type`.

Prototype

```
int dwarf_srcfrag_typeflag(
    Dwarf_SrcFrag       srcfrag,
    Dwarf_Flag*         ret_typeflag,
    Dwarf_Error*        error);
```

Parameters

srcfrag
Input. Input source fragment object.

ret_typeflag
Output. Source type flag associated with the given source fragment object.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

DW_DLV_OK
The source type flag associated with the source fragment object is returned.

DW_DLV_NO_ENTRY
Never returned.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- The given srcfrag is NULL.
- The given ret_typeflag is NULL.

dwarf_srcfrag_xreflist operation

The dwarf_srcfrag_xreflist operation retrieves the DW_TAG_IBM_xreflist DIE associated with the source fragment object.

Prototype

```
int dwarf_srcfrag_xreflist(
    Dwarf_SrcFrag       srcfrag,
    Dwarf_Die*          ret_die,
    Dwarf_Error*        error);
```

Parameters

srcfrag
Input. Input source fragment object.

ret_die
Output. The DW_TAG_IBM_xreflist DIE associated with the given source fragment object.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

DW_DLV_OK
The DW_TAG_IBM_xreflist DIE associated with the given source fragment object is returned.

DW_DLV_NO_ENTRY
No DW_TAG_IBM_xreflist DIE is associated with the given source fragment object.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- The given srcfrag is NULL.
- The given ret_die is NULL.
- There is cross reference information available, but the corresponding .debug_xref section is not found.
- The DW_TAG_IBM_xreflist DIE does not contain CU context information.

Cleanup

The DW_TAG_IBM_xreflist DIE returned by this API is owned by the source fragment object. You can not deallocate the returned list directly, but the source fragment object can be deallocated using dwarf_srcfrag_xref_dealloc().

dwarf_srcfrag_list_tags operation

The dwarf_srcfrag_list_tags operation looks at all the children DIEs under the DW_TAG_IBM_xreflist DIE associated with the given source fragment object, and returns a list of unique TAGs used by the children DIEs.

Prototype

```
int dwarf_srcfrag_list_tags(
    Dwarf_SrcFrag       srcfrag,
    Dwarf_Tag**         ret_taglist,
    Dwarf_Unsigned*     ret_n_taglist,
    Dwarf_Error*        error);
```

Parameters

srcfrag
Input. Source fragment object.

ret_taglist
Output. An array of DIE TAG values associated with the given source fragment object.

ret_n_taglist
Output. Number of DIE TAG values in the returned array of ret_taglist.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

DW_DLV_OK
The list of unique TAG value is returned.

DW_DLV_NO_ENTRY
No DW_TAG_IBM_xreflist DIE is associated with the given source fragment object.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- The given srcfrag is NULL.
- The given ret_dies or ret_n_dies is NULL.
- There is cross reference information available, but the corresponding .debug_xref section is not found.
- The DW_TAG_IBM_xreflist DIE does not contain CU context information.
- Can not allocate memory required to store the returned DW_TAG_IBM_xreflist_item DIEs in the persistent information.

Cleanup

The list of DIEs returned by this API is owned by the source fragment object. You can not deallocate the returned list directly, but the source fragment object can be deallocated using dwarf_srcfrag_xref_dealloc().

dwarf_srcfrag_list_items operation

The dwarf_srcfrag_list_items operation retrieves all the children DIEs of the given TAG value under the DW_TAG_IBM_xreflist DIE associated with the given source fragment object.

Prototype

```
int dwarf_srcfrag_list_items(
    Dwarf_SrcFrag       srcfrag,
    Dwarf_Half          tag,
    Dwarf_Die**         ret_dies,
    Dwarf_Unsigned*     ret_n_dies,
    Dwarf_Error*        error);
```

Parameters

srcfrag
Input. Input source fragment object.

tag
Input. The given tag value.

ret_dies
Output. An array of DIEs (with the given tag) associated with the given source fragment object.

ret_n_dies
Output. Number of DIEs in the returned array of ret_dies.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

DW_DLV_OK
The list of children DIEs of the given TAG value for the source fragment object is returned.

DW_DLV_NO_ENTRY

- No DW_TAG_IBM_xreflist DIE is associated with the given source fragment object.
- The DW_TAG_IBM_xreflist DIE does not have any children DIE matching the given tag.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- The given srcfrag is NULL.
- The given ret_dies or ret_n_dies is NULL.
- There is cross reference information available, but the corresponding .debug_xref section is not found.
- The DW_TAG_IBM_xreflist DIE does not contain CU context information.
- Cannot allocate memory required to store the returned DW_TAG_IBM_xreflist_item DIEs in the persistent information.

Cleanup

The list of DIEs returned by this API is owned by the source fragment object. You can not deallocate the returned list directly, but the source fragment object can be deallocated using dwarf_srcfrag_xref_dealloc().

dwarf_srcfrag_xref_dealloc operation

The dwarf_srcfrag_xref_dealloc operation deallocates internal storage held by a source fragment object to keep track of information about DW_TAG_IBM_xreflist DIE. If this API

succeeds, it invalidates all returned object(s) from these calls: `dwarf_srcfrag_xreflist()`,* `dwarf_srcfrag_list_tags()` , or `dwarf_srcfrag_list_items()`. If you are holding the returned object from these calls, do not make use of them after calling this API:

Prototype

```
int
  dwarf_srcfrag_xref_dealloc(
    Dwarf_SrcFrag       srcfrag,
    Dwarf_Error*        error);
```

Parameters

srcfrag
Input. Input source fragment object.

error
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

DW_DLV_OK
All internal storage held by the input source fragment object is deallocated.

DW_DLV_NO_ENTRY
Never

DW_DLV_ERROR
The given `srcfrag` is NULL:

Producer operations

The operations in this section create content in the `.debug_srcattr` debug section.

dwarf_srcattr_table operation

On first invocation of this API, `DW_TAG_IBM_src_attr` attribute is added to the given `DW_TAG_IBM_src_file` DIE. The value of the attribute contains the offset in .debug_srcattr containing the source attribute program. All subsequent producer APIs that adds row to a source attribute matrix will be added to this source attribute program until this API is called again with a different `DW_TAG_IBM_src_file` DIE.

Prototype

```
int dwarf_srcattr_table(
    Dwarf_P_Debug       dbg,
    Dwarf_P_Die         srcdie
    Dwarf_Error*        error);
```

Parameters

dbg
Input. This accepts the `Dwarf_P_Debug` object.

srcdie
Input. DIE to receive the `DW_AT_IBM_src_text` attribute. This should be a `DW_TAG_IBM_src_file` DIE.

error
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

DW_DLV_OK
All future .debug_srcattr matrix row additions will be applied to the source attribute program associated with the given source DIE.

DW_DLV_NO_ENTRY
Never returned.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- dbg is NULL.
- The Dwarf_P_Debug object contains invalid version information.
- srcdie is NULL.
- Cannot find the .debug_srcfiles section.
- Not enough memory to allocate internal objects.

dwarf_add_srcattr_entry operation

The dwarf_add_srcattr_entry operation adds a row into the source attribute matrix. The owner of the source attribute program is specified by the previous dwarf_srcattr_table() call. If a row has already been created with the same line_no and col_no, the existing source fragment object will be returned with the typeflag attribute merged with the existing entry. Additional information can be appended to the row via the returned source fragment object (Dwarf_P_SrcFrag). The rows entered into the source attribute matrix are always sorted using line_no first, then col_no.

Prototype

```
int dwarf_add_srcattr_entry(
    Dwarf_P_Debug       dbg,
    Dwarf_Unsigned      line_no,
    Dwarf_Signed        col_no,
    Dwarf_Flag          typeflag,
    Dwarf_P_SrcFrag*    ret_srcfrag,
    Dwarf_Error*        error);
```

Parameters

dbg
Input. This accepts the Dwarf_P_Debug object.

line_no
Input. An unsigned integer indicating a source line number where the source statement begins. Lines are numbered beginning at 1.

col_no
Input. A signed integer indicating a column number where the source statement begins. Columns are numbered beginning at 1. The value -1 indicates that this field is not used.

typeflag
Input. A flag indicating source type as defined in Dwarf_IBM_srcattr_type. The value 0 indicates that this field is not used.

ret_srcfrag
Output. Returned source fragment object representing this source attribute matrix row.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

DW_DLV_OK
A row has been entered successfully into the current source attribute program.

DW_DLV_NO_ENTRY
Never returned.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- dbg is NULL.
- The Dwarf_P_Debug object contains invalid version information.
- The value of line_no is not valid.
- Given ret_srcfrag is NULL.
- No current source attribute table is defined.
- Memory is not enough to allocate returned source fragment object.

dwarf_add_srcattr_xrefitem operation

The dwarf_add_srcattr_xrefitem operation adds a DIE to the given source fragment object. The input DIE must not have a parent DIE. The parent DIE is created during creation of the .debug_srcattr section, and the parent DIE will have the DW_TAG_IBM_xreflist tag. All the DIEs added to the input source fragment object are written into the .debug_xref section under a common DW_TAG_IBM_xreflist DIE.

Prototype

```
int dwarf_add_srcattr_xrefitem(
    Dwarf_P_Debug       dbg,
    Dwarf_P_SrcFrag     srcfrag,
    Dwarf_P_Die         xrefitem,
    Dwarf_Error*        error);
```

Parameters

dbg
Input. This accepts the Dwarf_P_Debug object.

srcfrag
Input. A source fragment object that is obtained from the dwarf_add_srcattr_entry call.

xrefitem
Input. DIE containing information about the source fragment object.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

DW_DLV_OK
The cross reference DIE is now associated with the given source fragment object.

DW_DLV_NO_ENTRY
Never returned.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- dbg is NULL.
- The Dwarf_P_Debug object contains invalid version information.
- Given srcfrag is NULL.

- Given xrefitem is NULL.
- No current source attribute table is defined.

dwarf_add_srcattr_altline operation

The dwarf_add_srcattr_altline operation adds an alternate line number to the given source fragment object.

Prototype

```
int dwarf_add_srcattr_altline(
    Dwarf_P_Debug       dbg,
    Dwarf_P_SrcFrag     srcfrag,
    Dwarf_Unsigned      altline_no,
    Dwarf_Error*        error);
```

Parameters

dbg
Input. This accepts the Dwarf_P_Debug object.

srcfrag
Input. A source fragment object that is obtained from the dwarf_add_srcattr_entry call.

altline_no
Input. An alternate line number for the source fragment object.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

DW_DLV_OK
The alternate line number is now associated with the given source fragment object.

DW_DLV_NO_ENTRY
Never returned.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- dbg is NULL.
- The Dwarf_P_Debug object contains invalid version information.
- The given srcfrag is NULL.
- No current source attribute table is defined.

dwarf_add_srcattr_relstmtno operation

The dwarf_add_srcattr_relstmtno operation adds a relative statement number to the given source fragment object.

Prototype

```
int dwarf_add_srcattr_relstmtno(
    Dwarf_P_Debug       dbg,
    Dwarf_P_SrcFrag     srcfrag,
    Dwarf_Unsigned      relstmtno,
    Dwarf_Error*        error);
```

Parameters

dbg
Input. This accepts the Dwarf_P_Debug object.

srcfrag
Input. A source fragment object that is obtained from the dwarf_add_srcattr_entry call.

relstmtno
Input. A relative statement number for the source fragment object.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

DW_DLV_OK
The relative statement number is now associated with the given source fragment object.

DW_DLV_NO_ENTRY
Never returned.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- dbg is NULL.
- The Dwarf_P_Debug object contains invalid version information.
- The given srcfrag is NULL.
- No current source attribute table is defined.

Chapter 10. DWARF expressions

The IBM extensions to DWARF expressions allow the DWARF expression evaluator to resolve generic expressions, in addition to those that specify a location or value. Because standard DWARF consumer operations do not cause an exception on overflow or underflow, this extension provides a DWARF stack-entity type for these expression operations. This means that floating point operations that cause exceptions will return error information.

In this document:

- DWARF operations are always discussed in terms of their effect on the DWARF stack machine.
- The input is discussed in terms of a stream of DWARF operations with their operands.

For specific information about standard DWARF expressions, refer to section 2.5 in *DWARF Debugging Information Format*, V4.

Defaults and general rules

The following defaults and general rules are associated with the addition of types to the stack machine:

- The default for arithmetic operations is unsigned 64-bit arithmetic.
- If a float or complex type is specified without a given size, then the element size defaults to 8 bytes.
- Bitwise operations on floating point types are not allowed.
- Const operations default to the type of the constant they are loading, when given in the op.

Operators

This section include operators that are introduced by the IBM extensions to DWARF expressions.

DW_OP_IBM_conv

The `DW_OP_IBM_conv` operation takes the next item on the stack and converts it from one type to another.

`DW_OP_IBM_conv` also takes a variable number of operands that are associated with the acquired stack item.

Notes:

- The first set of operands indicates the type of the value on the stack (the `from` type operand).
- The second set of operands indicates the new type (the `to` type operand).
- Both types will be encoded using the minimum amount of information required to define the type.
- The first element of the type description is an unsigned byte indicating the base type encoding; this is the same encoding that is used on the `DW_AT_encoding` attribute.
- The number of additional parameters expected is dependent on the base type.

Example

The code to convert a C unsigned short to an IEEE floating-point long double is:

```
DW_OP_IBM_conv DW_ATE_unsigned 2 DW_ATE_float 16
```

Parameters

Table 5. DW_OP_IBM_conv parameters

Base type encoding	Additional parameters
DW_ATE_signed_char DW_ATE_unsigned_char	No additional parameters.
DW_ATE_address DW_ATE_boolean DW_ATE_unsigned DW_ATE_signed DW_ATE_float DW_ATE_IBM_float_hex	**Container size** A 2-byte unsigned integer indicating the physical size of the type expressed in bytes. A value of 0xFFFF indicates a LEB128 value. An error will occur if 0xFFFF is given with a floating-point, Boolean or address type.
DW_ATE_numeric_string DW_ATE_signed_fixed DW_ATE_unsigned_fixed DW_ATE_packed_decimal DW_ATE_IBM_numeric_string_national	**decimal sign** a 1 byte value indicating the decimal sign encoding; this is the same encoding that is used on the `DW_AT_decimal_sign` attribute. If this does not apply, the value is zero. **digit count** a 1 byte unsigned value indicating the number of digits in an instance of the type. **decimal scale** a 1 byte signed value indicating the exponent of the base ten scale factor to be applied to an instance of the type. A scale of zero put the decimal point immediately to the right of the least significant digit. Positive scale moves the decimal point immediately 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. **container size** a LEB128 value indicating the physical size of the type expressed in bytes.

DW_OP_IBM_builtin

The `DW_OP_IBM_builtin` operation takes one unsigned-byte operand which indicates what kind of built-in function will occur.

Note: The `DW_OP_IBM` prefix indicates that an operation is a built-in function.

Built-in functions

Table 6. DW_OP_IBM_builtin functions

Sub Op	Description
DW_SubOP_builtin_strlen (0x01)	This Sub_0p treats the top item on the stack as a machine address (`Dwarf_Addr`) that refers to user storage. It then references the memory at that address and counts the number of bytes before a byte that contains the value 0x00 is encountered. Like `strlen` in the C library, the value 0x00 is not included in the count. The count is then placed on the stack as an 8-byte unsigned integer. A prefix operation `DW_OP_IBM_prefix` can be used to say that the address comes from local rather than user storage.
DW_SubOP_builtin_substr (0x02)	This Sub_0p takes the top three items from the stack: • A machine address (`Dwarf_Addr`) that refers to user storage • An 8-byte signed integer (`Dwarf_Signed`) that is the starting offset from the address • A signed 8-byte integer indicating the requested length of the substring If the substring has a negative length, then the substring length will extend until a byte containing the value 0x00 is encountered. The 0x00 byte will be part of the substring. The expression evaluator then allocates local memory space long enough for the given substring, and copies the string into the storage. Finally, the evaluator returns the address of the space on the stack as a `Dwarf_Addr` machine address. The allocated space will be in the local address space. A prefix operation `DW_OP_IBM_prefix` can be used to say that the address comes from local rather than user storage.
DW_SubOP_builtin_strcat (0x03)	This Sub_0p takes the top two items on the stack: • A machine address (`Dwarf_Addr`) that refers to user storage • An 8-byte signed integer (`Dwarf_Signed`) that is the starting offset from the address `DW_SubOP_builtin_strcat` treats them as machine addresses (`Dwarf_Addr`) in user storage. The API then behaves exactly like `strcat` in the ISO C library. The machine address of the local buffer is placed on the stack. A prefix operation `DW_OP_IBM_prefix` can be used to say that the incoming addresses come from local rather than user storage.

Table 6. DW_OP_IBM_builtin functions (continued)	
Sub Op	**Description**
DW_SubOP_builtin_pow (0x04)	This Sub_Op uses the top two values from the stack: • The base • The exponent The compiler returns the result of the base exponent to the stack. The result is in the same type as the base item unless a `DW_OP_IBM_prefix` is used.

DW_OP_IBM_prefix

The `DW_OP_IBM_prefix` operation allows the standard DWARF Expression Operations to encode items like long double float arithmetic.

`DW_OP_IBM_prefix` passes additional information to be used while the evaluator interprets the expression. `DW_OP_IBM_prefix` applies to the the next opcode that is a non-`DW_OP_IBM_prefix` opcode.

`DW_OP_IBM_prefix` takes at least two operands:

- The prefix type is a single unsigned byte that indicates the type of information is being provided
- Additional operands, with the number and size of each dependent on the prefix type

Additional parameters

The following table describes the currently supported prefix types and the operands that each requires.

Table 7. DW_OP_IBM_prefix additional parameters	
Prefix	**Description**

DW_SubOP_prefix_type(0x01)	This prefix has one additional parameter:
	Type
	A single unsigned byte indicating the DWARF base-type encoding. The following may not be specified on this prefix type: `DW_ATE_complex`, `DW_OP_IBM_user`, `DW_ATE_IBM_complex_hex`, `DW_ATE_IBM_packed_decimal` and `DW_ATE_IBM_zoned_decimal`
	Example: The following code would do an IEEE floating point add and uses the default floating point size:
	`DW_OP_IBM_prefix`
	`DW_SubOP_prefix_type`
	`DW_AT_float`
	`DW_OP_plus`

Table 7. DW_OP_IBM_prefix additional parameters (continued)

Prefix	Description

DW_SubOP_prefix_size(0x02)	This prefix has one additional parameter:
	Size
	Two unsigned bytes indicating the size of the type that is either the default or previously specified. A value of 0xFFFF indicates a LEB128 value. An error will occur if 0xFFFF is given with a floating point type. `DW_OP_IBM_user`, `DW_ATE_complex`, `DW_ATE_IBM_complex_hex`, `DW_ATE_IBM_packed_decimal` and `DW_ATE_IBM_zoned_decimal` may not be specified on this prefix type.
	Example: The following code would do a HEX long-double floating-point add:
	`DW_OP_IBM_prefix`
	`DW_SubOP_prefix_type`
	`DW_AT_IBM_float_hex`
	`DW_OP_IBM_prefix`
	`DW_SubOP_prefix_size 16`
	`DW_OP_plus`

Table 7. DW_OP_IBM_prefix additional parameters (continued)

Prefix	Description

DW_SubOP_prefix_kind(0x3)	This is a compressed prefix that passes all the type and size information at one time. It can be used for any type. The third and fourth parameters will normally be 0 for the basic types such as `char` or `float`. This prefix must be used for complex numbers, packed-decimal number, zoned decimal numbers, and user types. For user types, the sizes of the fields remain the same but their meanings are user defined.
	Type
	A 1-byte unsigned integer indicating the type. I uses the `DW_AT_encoding` types provided by DWARF. A 0xFFFF value indicates a LEB128 value. An error will occur if 0xFFFF is given with a floating point, Boolean or address type.
	Physical Size
	A 2-byte unsigned integer indicating the complete physical size of the instance in bytes. For a complex number this should include all parts. For a packed/ zoned decimal number it should include the sign bits and any padding.
	Logical Size/Element Size
	A 1-byte unsigned integer. For a complex number this is the size of each element. For a packed or zoned decimal number this is the number of digits. For any other type this should be 0x00.
	Decimal Places/Memory Space
	A 1-byte unsigned integer describing the number of digits after the implied period in a packed or zoned decimal number. For any other type, this should be 0x00. If this value is non-zero on an object of type `DW_ATE_address`, the address is in the local address space.
	Example: A long-double floating-point add could also be expressed as:
	`DW_OP_IBM_prefix`
	`DW_SubOP_prefix_kind`
	`DW_ATE_float 16 0 0`
	`DW_OP_PLUS`
	Example: Similarly, a HEX floating-point double complex number add would be:
	`DW_OP_IBM_prefix`
	`DW_SubOP_prefix_kind`
	`DW_ATE_IBM_complex_hex 16 8 0`
	`DW_OP_PLUS`

DW_SubOP_prefix_local_storage (0x04)	This prefix means that the address referenced by the following op is in local storage rather than user storage. There are no additional parameters.

DW_OP_IBM_logical_and

The DW_OP_IBM_logical_and operation takes the top two items on the stack and performs a logical and like in the ISO C library.

That is, it will place:

- An 8-byte integer 1 on the stack if both of the given stack values are not zero (in the appropriate type)
- An 8-byte integer 0 on the stack if either or both of the given stack entries are equal to zero

If the stack values are floating point, then they are first compared to a floating-point 0.

DW_OP_IBM_logical_or

The DW_OP_IBM_logical_or operation takes the top two items on the stack and performs a logical or like in the ISO C library.

That is, it will place:

- An 8-byte integer 1 on the stack if either of the given stack values are not zero
- An 8-byte integer 0 on the stack if both of the given stack entries are equal to zero

If the stack values are floating point, then they are first compared to a floating-point 0.

DW_OP_IBM_logical_not

The DW_OP_IBM_logical_not operation takes the top two items on the stack and performs a logical not like in the ISO C library.

That is, it will place:

- An 8-byte integer 1 on the stack if the given stack value is equal to zero (in the appropriate type)
- An 8-byte integer 0 on the stack if the given stack value is not equal to zero

If the stack values are floating point, then they are first compared to a floating-point 0.

DW_OP_IBM_user

The DW_OP_IBM_user operation indicates if the operation is a user-supplied function.

It takes a single unsigned byte to indicate which user operation is processed. User-supplied functions can either be unary or binary, depending on the type of function used to supply the function pointer. Unary functions use the top item on the stack, and binary functions use the top two items on the stack.

DW_OP_IBM_conjugate

The DW_OP_IBM_conjugate operation takes the top item on the stack and performs a complex conjugate operation. That is, it will reverses the sign of the imaginary part of the complex number and place the result on the stack.

DW_OP_IBM_wsa_addr

The DW_OP_IBM_wsa_addr operation takes no operand and pushes the WSA address on top of the stack.

DW_OP_IBM_loadmod_addr

The DW_OP_IBM_loadmod_addr operation takes no operand and pushes the start of the loadmodule address on top of the stack.

Location expression operations

The operations in this section are introduced by the IBM extensions to DWARF expressions.

dwarf_loclist_n operation

The dwarf_loclist_n operation decodes location list or location expression of a given attribute. It returns the location expressions as a list of Dwarf_Locdesc objects.

Prototype

```
int dwarf_loclist_n(
    Dwarf_Attribute     attr,
    Dwarf_Locdesc***    ret_llbuf,
    Dwarf_Signed *      ret_listlen,
    Dwarf_Error*        error);
```

Parameters

attr
Input. DWARF attribute holding a location list or location expression.

ret_llbuf
Output. An array of Dwarf_Locdesc* objects.

ret_listlen
Output. Number of Dwarf_Locdesc* objects in the array.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

DW_DLV_OK
An array of Dwarf_Locdesc* object is returned.

DW_DLV_NO_ENTRY
Never returned.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- The given attr is NULL.
- The given ret_llbuf or ret_listlen is NULL.
- The form of the attribute is not supported.
- Unable to allocate memory for creating internal objects.

Cleanups

```
Dwarf_Locdesc** loclist;
Dwarf_Signed    loclist_n;

dwarf_loclist_n (attr, &loclist, &loclist_n, &err);

for (i=0; i<loclist_n; i++) {
  dwarf_dealloc (dbg, loclist[i]->ld_s, DW_DLA_LOC_BLOCK);
  dwarf_dealloc (dbg, loclist[i], DW_DLA_LOCDESC);
}
dwarf_dealloc (dbg, loclist, DW_DLA_LIST);
```

dwarf_get_loc_list_given_offset operation

The `dwarf_get_loc_list_given_offset` operation decodes location list given an offset within `.debug_loc`. The offset must point to the beginning of a location list. The order of expression locations returned is in the same order as the encoded information in `.debug_loc`.

Prototype

```
int dwarf_get_loc_list_given_offset (
    Dwarf_Debug         dbg,
    Dwarf_Off           offset,
    Dwarf_Locdesc***    ret_llbuf,
    Dwarf_Signed *      ret_listlen,
    Dwarf_Off*          ret_nextoff,
    Dwarf_Error*        error);
```

Parameters

dbg
Input. `libdwarf` consumer instance.

offset
Input. The offset to the beginning of the location list.

ret_llbuf
Output. An array of `Dwarf_Locdesc*` objects.

ret_listlen
Output. Number of `Dwarf_Locdesc*` objects in the array.

ret_nextoff
Output. The offset to the beginning of the next location list. This field can be NULL, in which case, this value will not be used.

error
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

DW_DLV_OK
An array of `Dwarf_Locdesc*` object is returned.

DW_DLV_NO_ENTRY

- `.debug_loc` debug section does not exist or is empty.
- `.debug_info` debug section does not exist or is empty.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- The given dbg is NULL.
- Unable to determine the offset of the next location list entry.
- Unable to allocate memory for creating internal objects.

Cleanups

```
Dwarf_Locdesc** loclist;
Dwarf_Signed    loclist_n;

dwarf_get_loc_list_given_offset (dbg, offset, &loclist, &loclist_n, NULL, &err);

for (i=0; i<loclist_n; i++) {
  dwarf_dealloc (dbg, loclist[i]->ld_s, DW_DLA_LOC_BLOCK);
  dwarf_dealloc (dbg, loclist[i], DW_DLA_LOCDESC);
}
dwarf_dealloc (dbg, loclist, DW_DLA_LIST);
```

Chapter 11. DWARF library debugging facilities

These consumer APIs can be used when debugging a DWARF application.

Machine-register name API

These APIs provide specific information about a register used within the location expression.

Debug sections

IBM has created an extension to the DWARF sections and Debug Information Entries (DIEs). Only the .debug_info section describes the contents and usage of a machine register.

DW_FRAME_390_REG_type object

The machine registers are accessed through the DW_FRAME_390_REG_type data structure. This type is transparent, machine-dependent and describes the z/OS CPU-register assignments.

Type definition

```
typedef enum {
   DW_FRAME_390_gpr0         = 0,
   DW_FRAME_390_gpr1         = 1,
   DW_FRAME_390_gpr2         = 2,
   DW_FRAME_390_gpr3         = 3,
   DW_FRAME_390_gpr4         = 4,
   DW_FRAME_390_gpr5         = 5,
   DW_FRAME_390_gpr6         = 6,
   DW_FRAME_390_gpr7         = 7,
   DW_FRAME_390_gpr8         = 8,
   DW_FRAME_390_gpr9         = 9,
   DW_FRAME_390_gpr10        = 10,
   DW_FRAME_390_gpr11        = 11,
   DW_FRAME_390_gpr12        = 12,
   DW_FRAME_390_gpr13        = 13,
   DW_FRAME_390_gpr14        = 14,
   DW_FRAME_390_gpr15        = 15,
   DW_FRAME_390_fpr0         = 16,
   DW_FRAME_390_vr0          = 16,
   DW_FRAME_390_fpr2         = 17,
   DW_FRAME_390_vr2          = 17,
   DW_FRAME_390_fpr4         = 18,
   DW_FRAME_390_vr4          = 18,
   DW_FRAME_390_fpr6         = 19,
   DW_FRAME_390_vr6          = 19,
   DW_FRAME_390_fpr1         = 20,
   DW_FRAME_390_vr1          = 20,
   DW_FRAME_390_fpr3         = 21,
   DW_FRAME_390_vr3          = 21,
   DW_FRAME_390_fpr5         = 22,
   DW_FRAME_390_vr5          = 22,
   DW_FRAME_390_fpr7         = 23,
   DW_FRAME_390_vr7          = 23,
   DW_FRAME_390_fpr8         = 24,
   DW_FRAME_390_vr8          = 24,
   DW_FRAME_390_fpr10        = 25,
   DW_FRAME_390_vr10         = 25,
   DW_FRAME_390_fpr12        = 26,
   DW_FRAME_390_vr12         = 26,
   DW_FRAME_390_fpr14        = 27,
   DW_FRAME_390_vr14         = 27,
   DW_FRAME_390_fpr9         = 28,
   DW_FRAME_390_vr9          = 28,
   DW_FRAME_390_fpr11        = 29,
   DW_FRAME_390_vr11         = 29,
   DW_FRAME_390_fpr13        = 30,
   DW_FRAME_390_vr13         = 30,
   DW_FRAME_390_fpr15        = 31,
   DW_FRAME_390_vr15         = 31,
   DW_FRAME_390_cr0          = 32,
```

```
    DW_FRAME_390_cr1          = 33,
    DW_FRAME_390_cr2          = 34,
    DW_FRAME_390_cr3          = 35,
    DW_FRAME_390_cr4          = 36,
    DW_FRAME_390_cr5          = 37,
    DW_FRAME_390_cr6          = 38,
    DW_FRAME_390_cr7          = 39,
    DW_FRAME_390_cr8          = 40,
    DW_FRAME_390_cr9          = 41,
    DW_FRAME_390_cr10         = 42,
    DW_FRAME_390_cr11         = 43,
    DW_FRAME_390_cr12         = 44,
    DW_FRAME_390_cr13         = 45,
    DW_FRAME_390_cr14         = 46,
    DW_FRAME_390_cr15         = 47,
    DW_FRAME_390_ar0          = 48,
    DW_FRAME_390_ar1          = 49,
    DW_FRAME_390_ar2          = 50,
    DW_FRAME_390_ar3          = 51,
    DW_FRAME_390_ar4          = 52,
    DW_FRAME_390_ar5          = 53,
    DW_FRAME_390_ar6          = 54,
    DW_FRAME_390_ar7          = 55,
    DW_FRAME_390_ar8          = 56,
    DW_FRAME_390_ar9          = 57,
    DW_FRAME_390_ar10         = 58,
    DW_FRAME_390_ar11         = 59,
    DW_FRAME_390_ar12         = 60,
    DW_FRAME_390_ar13         = 61,
    DW_FRAME_390_ar14         = 62,
    DW_FRAME_390_ar15         = 63,
    DW_FRAME_390_PSW_mask     = 64,
    DW_FRAME_390_PSW_address  = 65,
    DW_FRAME_390_WSA_address  = 66,  /* DEPRECATED */
    DW_FRAME_390_loadmodule   = 67,  /* DEPRECATED */
    DW_FRAME_390_CEESTART     = 67,  /* DEPRECATED */
    DW_FRAME_390_vr16         = 68,
    DW_FRAME_390_vr18         = 69,
    DW_FRAME_390_vr20         = 70,
    DW_FRAME_390_vr22         = 71,
    DW_FRAME_390_vr17         = 72,
    DW_FRAME_390_vr19         = 73,
    DW_FRAME_390_vr21         = 74,
    DW_FRAME_390_vr23         = 75,
    DW_FRAME_390_vr24         = 76,
    DW_FRAME_390_vr26         = 77,
    DW_FRAME_390_vr28         = 78,
    DW_FRAME_390_vr30         = 79,
    DW_FRAME_390_vr25         = 80,
    DW_FRAME_390_vr27         = 81,
    DW_FRAME_390_vr29         = 82,
    DW_FRAME_390_vr31         = 83,
    DW_FRAME_390_LAST_REG_NUM
  } DW_FRAME_390_REG_type;
```

Members

The members of DW_FRAME_390_REG_type are organized as follows:

DW_FRAME_390_gpr0 to DW_FRAME_390_gpr15
General-purpose registers.

DW_FRAME_390_fpr0 to DW_FRAME_390_fpr15
Floating-point registers.

DW_FRAME_390_cr0 to DW_FRAME_390_cr15
Control registers.

DW_FRAME_390_ar0 to DW_FRAME_390_ar15
Address registers.

DW_FRAME_390_PSW_mask
PSW mask.

DW_FRAME_390_PSW_address
PSW address.

DW_FRAME_390_WSA_address
WSA address.

DW_FRAME_390_loadmodule to DW_FRAME_390_CEESTART
Load-module address.

DW_FRAME_390_vr0 to DW_FRAME_390_vr31
Vector registers.

DW_FRAME_390_LAST_REG_NUM
The number of columns in the Frame Table.

dwarf_register_name operation

The `dwarf_register_name` operation queries the name of the given machine register.

Prototype

```
int dwarf_register_name(
  Dwarf_Debug           dbg,
  Dwarf_Signed          reg,
  char**                ret_name,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a `libdwarf` consumer object.

reg
Input. This accepts the machine-register number.

ret_name
Output. This returns the register name.

error
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

The `dwarf_register_name` operation returns `DW_DLV_NO_ENTRY` if `reg` is not a valid register number.

Relocation type name consumer API

This API provides specific information about a relocation type.

Relocation macros

The following relocation macros are defined for the z/OS operating system.

R_390_NONE
Value = 0. No relocation.

R_390_8
Value = 1. Direct 8-bit.

R_390_12
Value = 2. Direct 12-bit.

R_390_16
Value = 3. Direct 16-bit.

R_390_32
Value = 4. Direct 32-bit.

R_390_PC32
Value = 5. PC-relative 32-bit.

R_390_GOT12
Value = 6. 12-bit GOT entry.

R_390_GOT32
Value = 7. 32-bit GOT entry.

R_390_PLT32
Value = 8. 32-bit PLT entry.

R_390_COPY
Value = 9. Copy symbol at run time.

R_390_GLOB_DAT
Value = 10. Create GOT entry.

R_390_JMP_SLOT
Value - 11. Create PLT entry.

R_390_RELATIVE
Value = 12. Adjust by program base.

R_390_GOTOFF
Value = 13. 32-bit offset to GOT.

R_390_GOTPC
Value = 14. 32-bit PC-relative offset to GOT.

R_390_GOT16
Value = 15. 16-bit GOT entry.

R_390_PC16
Value = 16. PC-relative 16-bit.

R_390_PC16DBL
Value = 17. PC-relative 16-bit redirected to 1.

R_390_PLT16DBL
Value = 18. 16-bit redirected to 1 PLT entry.

R_390_PC32DBL
Value = 19. PC relative 32-bit redirected to 1.

R_390_PLT32DBL
Value = 20. 32-bit redirected to 1 PLT entry.

R_390_GOTPCDBL
Value = 21. 32-bit redirected to 1 PC-relative offset to GOT.

R_390_64
Value = 22. Direct 64-bit.

R_390_PC64
Value = 23. PC relative 64-bit.

R_390_GOT64
Value = 24. 64-bit GOT entry.

R_390_PLT64
Value = 25. 64-bit PLT entry.

R_390_GOTENT
Value = 26. 32-bit redirected to 1 PC-relative GOT entry.

R_390_NUM
Value = 27. Number of defined types.

dwarf_reloc_type_name operation

The dwarf_reloc_type_name operation queries the name of the given relocation type.

Prototype

```
int dwarf_reloc_type_name(
  Dwarf_Debug           dbg,
  Dwarf_Signed          reloc_type,
  char**                ret_name,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer object.

reloc_type
Input. This accepts one of the relocation macros, as defined in "Relocation macros" on page 143.

ret_name
Output. This returns the relocation-type name.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_reloc_type_name operation returns DW_DLV_NO_ENTRY if reloc_type is not a valid relocation type.

Utility consumer operations

These utilities assist in debugging a program-analysis tool that is being developed.

dwarf_build_version operation

This operation displays the build ID of the dwarf library. Every release/PTF of the dwarf library will have an unique build ID. This information is useful for providing service information to IBM customer support. Calling this function will emit the build ID string (encoded in ISO8859-1) to stdout.

Prototype

```
char*
  dwarf_build_version (void);
```

Return values

Returns build ID of the dwarf library. The returned string is encoded in ISO8859-1.

Example

```
/* Compile this code with ASCII option */
printf ("Library(dwarf) Level(%s)\n", dwarf_build_version());
```

dwarf_show_error operation

If the user error handler is responsible for the error display, then the dwarf_show_error operation enables or disables the verbose display.

The verbose display is disabled by default. Enabling the display will send the message number, text and any available traceback to STDERR.

Prototype

```
int dwarf_show_error (
  Dwarf_Debug           dbg,
  Dwarf_Bool            new_show,
  Dwarf_Bool*           ret_prev_show,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer object.

new_show
Input. This accepts the Boolean value that will enable or disable the verbose error display.

ret_prev_show
Output. This returns the previous Boolean value replaced by the new_show value.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_show_error operation never returns DW_DLV_NO_ENTRY.

Memory allocation

There is no storage to deallocate.

dwarf_set_stringcheck operation

The dwarf_set_stringcheck operation enables or disables the libdwarf internal string checks.

This API must be called before a Dwarf_Debug object is created for it to have an effect.

Internal string checks ensure that the string literals have a proper length and are within the bounds of the debug section. String checks are done when libdwarf operations retrieve string literals from the debug information. By default, string checks are enabled. This is the safest way to run your application. If disabled, then performance will improve.

The previous setting is returned when the operation has finished.

Prototype

```
int dwarf_set_stringcheck(
  int                   stringcheck);
```

Parameters

stringcheck
Input. This accepts 0 to enable the checks, and 1 to disable them.

Return values

The dwarf_set_stringcheck operation never returns DW_DLV_NO_ENTRY.

Memory allocation

There is no storage to deallocate.

Chapter 12. Producer APIs for standard DWARF sections

These are IBM's extended producer operations for the standard DWARF sections.

Initialization and termination producer operations

The operations that create, terminate, and specify the codeset of DWARF producer objects.

dwarf_producer_target operation

This operation sets up the size of the pointers and relocation types within the producer DWARF object using the information provided in the ELF file header.

Prototype

```
int dwarf_producer_target(
     Dwarf_P_Debug          dbg,
     Elf*                   elfptr,
     Dwarf_Error*           error);
```

Parameters

dbg
Input. This accepts a `libdwarf` producer object.

elfptr
Input. This accepts an ELF descriptor.

error
Input/Output. This accepts or returns the `Dwarf_Error` object.

Return values

DW_DLV_OK
Returned upon successful completion of the operation.

DW_DLV_NO_ENTRY
Never returned.

DW_DLV_ERROR
Returned if:

- `dbg` is NULL
- `elfptr` is NULL
- Header information within the given ELF descriptor is corrupt

dwarf_producer_write_elf operation

This operation writes the contents of the ELF descriptor to the side file.

This content includes:

- The ELF file header, section headers and section data
- Generated ELF sections
- Sections, such as `.debug_info`, generated via `libdwarf` operations

The section data is retrieved via the dwarf_get_section_bytes operation, which also sets the final section data length. The data must be in the exact order of the ELF-section index values. These values are assigned by calls to the callback function passed to either the dwarf_producer_init or dwarf_producer_init_b operation.

User ELF sections, such as .text and .data, are not generated via libdwarf operations. The section header must be complete, and include the section data length. user_elf_data may be NULL if all the user sections are SHT_NOBITS. ELF-section index values will follow those in the generated list.

Prototype

```
int dwarf_producer_write_elf(
     Dwarf_P_Debug         dbg,
     Elf*                  elfptr,
     int                   n_gend_scns,
     Elf_Scn **            gend_elf_scns,
     char **               gend_elf_names,
     int                   n_user_scns,
     Elf_Scn **            user_elf_scns,
     char **               user_elf_names,
     char **               user_elf_data,
     Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

elfptr
Input. This accepts the ELF descriptor.

n_gend_scns
Input. This accepts the number of generated ELF sections.

gend_elf_scns,
Input. This accepts the generated ELF sections.

gend_elf_names
Input. This accepts the name of the generated ELF section.

n_user_scns
Input. This accepts the number of user ELF sections.

user_elf_scns
Input. This accepts the user ELF section.

user_elf_names
Input. This accepts the name of the user ELF section.

user_elf_data
Input. This accepts the section data of the user ELF section.

error
Input/Output. This accepts or returns the Dwarf_Error object.

Return values

DW_DLV_OK
Returned upon successful completion of the operation.

DW_DLV_NO_ENTRY
Never returned.

DW_DLV_ERROR
Returned if:

- dbg is NULL.
- elfptr is NULL.

dwarf_p_set_codeset operation

This operation specifies the code set for all the strings (character arrays) that will be passed into the `libdwarf` producer operations.

Prototype

```
int dwarf_p_set_codeset(
   Dwarf_P_Debug    dbg,
   const __ccsid_t  codeset_id,
   __ccsid_t*       prev_cs_id,
   Dwarf_Error*     error);
```

Parameters

dbg
Input. This accepts the `Dwarf_P_Debug` object.

codeset_id
This accepts the codeset for all the strings that will be passed into the `libdwarf` producer operations. You can obtain this ID by calling `__toCcsid()`. For more information on the `__toCcsid()` function, see the library functions in *z/OS C/C++ Run-Time Library Reference*. For a list of codesets that are supported, see *z/OS C/C++ Programming Guide*.

prev_cs_id
Output. This returns the code set that was specified in the last call to this operation. If the operation is called for the first time, this returns ISO8859-1, which is the default code set. If you specify NULL, then the previously specified codeset will not be returned.

error
Input/Output. This accepts and returns the `Ddpi_Error` object. This is a required parameter that handles error information generated by the producer or consumer application. If `error` is not NULL, then error information will be stored in the given object. If `error` is NULL, then the `libddpi` error process will look for an error-handling callback function that was specified by the `ddpi_init` operation. If no callback function was specified, then the error process will abort.

Return values

DW_DLV_OK
Returned upon successful completion of the operation.

DW_DLV_NO_ENTRY
Never returned.

DW_DLV_ERROR
Returned if:

- dbg is NULL.
- `codeset_id` is invalid.
- `dwarf_p_set_codeset` is unable to convert the specified codeset to an internal codeset.

dwarf_error-information producer operations

This section discusses the set of operations that manipulate the error objects for producers.

dwarf_p_seterrhand operation

The dwarf_p_seterrhand operation assigns a new error handler to the producer error object.

Prototype

```
Dwarf_Handler dwarf_p_seterrhand(
    Dwarf_P_Debug         dbg,
    Dwarf_Handler         errhand);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

errhand
Input. This accepts the error handler or NULL.

Return values

DW_DLV_OK
Returned upon successful completion of the operation.

DW_DLV_NO_ENTRY
Never returned.

DW_DLV_ERROR
Returned if dbg is NULL.

dwarf_p_seterrarg operation

The dwarf_p_seterrarg operation assigns a new error argument to the producer error object.

Prototype

```
Dwarf_Ptr dwarf_p_seterrarg(
    Dwarf_P_Debug         dbg,
    Dwarf_Ptr             errarg);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

errhand
Input. This accepts the error invocation-ID argument.

Return values

DW_DLV_OK
Returned with the previous error argument upon successful completion of the operation.

DW_DLV_NO_ENTRY
Never returned.

DW_DLV_ERROR
Returned if dbg is NULL.

dwarf_p_show_error operation

The dwarf_p_show_error operation enables or disables the verbose error display.

The default is false, when the user error handler is responsible for the error display. When set to true, messages are sent to STDERR when an error is detected, showing the message number, text and available traceback.

Prototype

```
int dwarf_p_show_error(
    Dwarf_P_Debug         dbg,
    Dwarf_Bool            new_show,
    Dwarf_Bool*           ret_prev_show,
    Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

new_show
Input. This accepts the flag that indicates whether or not to display the error.

ret_prev_show
Input. This accepts the flag that indicates whether or not to display the previous setting that is returned.

error
Input/Output. This accepts or returns the Dwarf_Error object.

Return values

DW_DLV_OK
Returned upon successful completion of the operation.

DW_DLV_NO_ENTRY
Never returned.

DW_DLV_ERROR
Returned if:

- dbg is NULL.
- ret_prev_show is NULL.

Chapter 13. Debug-section creation and termination operations

These APIs deal with creating and terminating debug sections within the ELF object.

dwarf_add_section_to_debug operation

The dwarf_add_section_to_debug operation creates a new debug section on an initial call.

If a section already exists, then dwarf_add_section_to_debug creates a separate instance of the section (with a separate unit header).

Prototype

```
int dwarf_add_section_to_debug(
    Dwarf_P_Debug         dbg,
    char *                section_name,
    Dwarf_P_Section*      ret_section,
    Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

section_name
Input. This accepts the debug section name.

ret_section
Output. This returns the Dwarf_P_Section.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

DW_DLV_OK
Returned upon successful completion of the operation.

DW_DLV_NO_ENTRY
Never returned.

DW_DLV_ERROR
Returned if:

- dbg is NULL
- Debug section name is NULL
- Returned section object is NULL

dwarf_section_finish operation

The dwarf_section_finish operation completes a debug section, after which no more information can be added.

Prototype

```
int dwarf_section_finish(
    Dwarf_P_Debug         dbg,
```

```
    Dwarf_P_Section       section,
    Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a `libdwarf` producer object.

section
Input. This accepts the `Dwarf_P_Section`.

error
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

DW_DLV_OK
Returned upon successful completion of the operation.

DW_DLV_NO_ENTRY
Never returned.

DW_DLV_ERROR
Returned if:

- dbg is NULL
- section object given is NULL
- section object given has been completed before (in other words, `dwarf_section_finish` has been called before for this object)

Chapter 14. ELF section operations

These operations are used for creating and querying information on other sections in ELF that are not part of the debug section. Examples of these sections are `.strtab` (string table) and `.symtab` (symbol table).

dwarf_elf_create_string operation

The `dwarf_elf_create_string` operation creates an entry in the `.strtab` section.

Only one entry is created for a given string, therefore this operation can be used to look up the index of a given string.

Prototype

```
int dwarf_elf_create_string(
    Dwarf_P_Debug         dbg,
    char*                 string,
    Dwarf_Unsigned*       ret_elf_stridx,
    Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a `libdwarf` producer object.

string
Input. This accepts the ELF string (NULL terminated).

ret_elf_stridx
Output. This returns the ELF `strtab` index.

error
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

The `dwarf_elf_create_string` operation returns:

- `DW_DLV_OK` if successful
- `DW_DLV_ERROR` if:
  - dbg is NULL
  - string is NULL
  - Returned parameter is NULL

`dwarf_elf_create_string` never returns `DW_DLV_NO_ENTRY`.

dwarf_elf_create_symbol operation

The `dwarf_elf_create_symbol` operation creates an ELF symbol in `.symtab`.

Prototype

```
int dwarf_elf_create_symbol(
    Dwarf_P_Debug         dbg,
    char*                 sym_name,
    Dwarf_Addr            sym_value,
    Dwarf_Unsigned        sym_size,
    unsigned char         sym_type,
    unsigned char         sym_bind,
    unsigned char         sym_other,
```

```
    Dwarf_Signed          sym_shndx,
    Dwarf_Unsigned*       ret_elf_symidx,
    Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

sym_name
Input. This accepts the ELF symbol name.

sym_value
Input. This accepts the ELF symbol value.

sym_size
Input. This accepts the ELF symbol size.

sym_type
Input. This accepts the ELF symbol type.

sym_bind
Input. This accepts the ELF symbol bind.

sym_other
Input. This accepts the ELF symbol other.

sym_shndx
Input. This accepts the ELF section idx.

ret_elf_stridx
Output. This returns the ELF .symtab index.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_elf_create_symbol operation returns:

- DW_DLV_OK if successful
- DW_DLV_ERROR if:
  - dbg is NULL
  - sym_name is NULL
  - Returned parameter is NULL

dwarf_elf_create_symbol never returns DW_DLV_NO_ENTRY.

dwarf_elf_producer_symbol_index_list operation

The dwarf_elf_producer_symbol_index_list operation retrieves the ELF symbol table-entry index, given a symbol name.

Prototype

```
int dwarf_elf_producer_symbol_index_list(
    Dwarf_P_Debug         dbg,
    char*                 sym_name,
    Dwarf_Unsigned**      ret_elf_symlist,
    Dwarf_Unsigned*       ret_elf_symcnt,
    Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

sym_name
Input. This accepts the ELF symbol name.

ret_elf_symlist
Output. This returns a list of ELF symbol indexes for the given name.

ret_elf_symcnt
Output. This returns the number of ELF symbol indexes in the list.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_elf_producer_symbol_index_list operation returns:

- DW_DLV_OK if successful
- DW_DLV_ERROR if:
  - dbg is NULL
  - sym_name is NULL
  - Returned parameters are NULL

dwarf_elf_producer_symbol_index_list returns DW_DLV_NO_ENTRY if either .symtab is not found or if sym_name is not found in .symtab.

Memory allocation

You can deallocate the parameters as required.

Example: The following example is a code fragment that deallocates the ret_elf_symilst parameter:

```
if (dwarf_elf_producer_symbol_index_list(dbg, ..., &ret_elf_symilst,
                                         &ret_elf_symcnt, &err)
     == DW_DLV_OK)
  {  dwarf_p_dealloc (dbg, ret_elf_symilst, DW_DLA_LIST); }
```

dwarf_elf_producer_string operation

The dwarf_elf_producer_string operation retrieves the ELF string table entry data for a given .strtab index.

Prototype

```
int dwarf_elf_producer_string(
    Dwarf_P_Debug         dbg,
    Dwarf_Unsigned        elf_stridx,
    char**                ret_str_name,
    Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

elf_stridx
Input. This accepts the ELF strtab index.

ret_str_name
Output. This returns the ELF string name.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_elf_producer_string operation returns:

- DW_DLV_OK if successful
- DW_DLV_ERROR if:
  - dbg is NULL
  - Returned parameter is NULL

dwarf_elf_producer_string returns DW_DLV_NO_ENTRY if either .symtab is not found or if elf_stridx is out of bounds.

dwarf_elf_producer_symbol operation

The dwarf_elf_producer_symbol operation retrieves the ELF symbol for a given .strtab index.

Prototype

```
int dwarf_elf_producer_symbol(
    Dwarf_P_Debug         dbg,
    Dwarf_Unsigned        elf_symidx,
    char**                ret_sym_name,
    Dwarf_Addr*           ret_sym_value,
    Dwarf_Unsigned*       ret_sym_size,
    unsigned char*        ret_sym_type,
    unsigned char*        ret_sym_bind,
    unsigned char*        ret_sym_other,
    Dwarf_Signed*         ret_sym_shndx,
    Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

elf_symidx
Input. This accepts the ELF symbol table (.symtab) index.

ret_sym_name
Output. This returns the ELF symbol name.

ret_sym_value
Output. This returns the ELF symbol value.

ret_sym_size
Output. This returns the ELF symbol size.

ret_sym_type
Output. This returns the ELF symbol type.

ret_sym_bind
Output. This returns the ELF symbol bind.

ret_sym_other
Output. This returns the ELF symbol other.

ret_sym_shndx
Output. This returns the ELF section idx.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_elf_producer_string operation returns:

- DW_DLV_OK if successful
- DW_DLV_ERROR if:
  - dbg is NULL
  - Returned parameter is NULL

dwarf_elf_producer_symbol returns DW_DLV_NO_ENTRY if either .symtab is not found or if elf_symidx is out of bounds.

dwarf_elf_create_section_hdr_string operation

The dwarf_elf_create_section_hdr_string operation creates an entry in the ELF section-header string table (.shstrtab).

Only one entry is created for each given string. Therefore, it can also be used to look up the index of a given string.

Prototype

```
int dwarf_elf_create_section_hdr_string(
    Dwarf_P_Debug         dbg,
    char*                 string,
    Dwarf_Unsigned*       ret_elf_hstridx,
    Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

string
Input. This accepts the ELF string (NULL terminated).

ret_elf_hstridx
Output. This returns the ELF shstrtab index.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_elf_create_section_hdr_string API returns:

- DW_DLV_OK if successful
- DW_DLV_ERROR if:
  - dbg is NULL
  - string is NULL
  - Returned parameter is NULL

dwarf_elf_create_section_hdr_string never returns DW_DLV_NO_ENTRY.

dwarf_elf_producer_section_hdr_string

The dwarf_elf_producer_section_hdr_string operation retrieves the entry data in the string table of the ELF section header, by index.

Prototype

```
int dwarf_elf_producer_section_hdr_string(
    Dwarf_P_Debug         dbg,
    Dwarf_Unsigned        elf_hstridx,
    char**                ret_str_name,
    Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

elf_hstridx
This accepts the ELF shstrtab index.

ret_str_name
Output. This returns the ELF string name.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_elf_producer_section_hdr_string API returns:

- DW_DLV_OK if successful
- DW_DLV_ERROR if:
  - dbg is NULL
  - Returned parameter is NULL

dwarf_elf_producer_section_hdr_string returns DW_DLV_NO_ENTRY if either .symtab is not found or if elf_hstridx is out of bounds.

Chapter 15. DIE creation and modification operations

These operations are used to create DIEs in DIE sections, and to add attributes of different forms to the DIEs.

dwarf_add_die_to_debug_section operation

The dwarf_add_die_to_debug_section operation attaches a DIE in an arbitrary DIE-format debug section as root.

Prototype

```
int dwarf_add_die_to_debug_section(
    Dwarf_P_Debug         dbg,
    Dwarf_P_Section       section,
    Dwarf_P_Die           first_die,
    Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

section
Input. This accepts the owning Dwarf_P_Section.

first_die
Input. This accepts the first (root) DIE in the section.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_add_die_to_debug_section operation returns:

- DW_DLV_OK if successful
- DW_DLV_ERROR if:
  - dbg is NULL
  - section object is NULL
  - section object has been completed
  - Given root DIE is NULL
  - The tag of the root DIE does not match DW_TAG_compile_unit or DW_TAG_partial_unit

dwarf_add_die_to_debug_section never returns DW_DLV_NO_ENTRY.

dwarf_add_AT_block_const_attr operation

The dwarf_add_AT_block_const_attr operation adds an arbitrary attribute to the specified DIE and encodes the value using the form of block class.

Prototype

```
Dwarf_P_Attribute
    dwarf_add_AT_block_const_attr(
    Dwarf_P_Die         ownerdie,
    Dwarf_Half          attr,
    Dwarf_Unsigned      block_size,
```

```
    Dwarf_Ptr           block_data,
    Dwarf_Error*        error);
```

Parameters

ownerdie
Input. This accepts the DIE that receives the given attribute.

attr
Input. This accepts the attribute name.

block_size
Input. This accepts the block data in a fixed sized buffer.

block_data
Input. This accepts the length of block data buffer.

error
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

The `dwarf_add_AT_block_const_attr` operation returns the `Dwarf_P_Attribute` descriptor for `attr` on success, and `DW_DLV_BADADDR` if:

- The `ownerdie` object is NULL
- The `ownerdie` object does not have a valid producer debug instance
- The memory to allocate internal objects is not adequate

dwarf_add_AT_const_value_block operation

The `dwarf_add_AT_const_value_block` operation adds the `DW_AT_const_value` attribute to the specified DIE and encodes the value using the form of block class.

Prototype

```
Dwarf_P_Attribute
    dwarf_add_AT_const_value_block(
    Dwarf_P_Die         ownerdie,
    Dwarf_Unsigned      block_size,
    Dwarf_Ptr           block_data,
    Dwarf_Error*        error);
```

Parameters

ownerdie
Input. This accepts the DIE that receives the given attribute.

block_size
Input. This accepts the constant value data in a fixed-size buffer.

block_data
Input. This accepts the length of constant value data buffer.

error
Input/output. This accepts or returns the `Dwarf_Error` object.

Return values

The `dwarf_add_AT_const_value_block` operation returns the `Dwarf_P_Attribute` descriptor for `attr` on success, and `DW_DLV_BADADDR` if:

- The `ownerdie` object is NULL

- The ownerdie object does not have a valid producer debug instance
- The memory to allocate internal objects is not adequate

dwarf_add_AT_reference__noninfo_with_reloc operation

The dwarf_add_AT_reference_noninfo_with_reloc operation adds references to DIE that does not belong to the .debug_info section.

This type of reference (DW_FORM_sec_offset) is an offset from the beginning of the debug section of other DIEs. The offset field is 4 bytes for 32-bit objects, and 8 bytes for 64-bit objects.

Prototype

```
Dwarf_P_Attribute dwarf_add_AT_reference_noninfo_with_reloc (
    Dwarf_P_Debug       dbg,
    Dwarf_P_Die         ownerdie,
    Dwarf_Half          attr,
    Dwarf_P_Die         otherdie,
    Dwarf_Error*        error);
```

Parameters

dbg
Input. This accept the Dwarf_P_Debug object.

ownerdie
Input. DIE to receive the given attribute.

attr
Input. DIE attribute name.

otherdie
Input. DIE being referenced by this attribute.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_add_AT_reference_noninfo_with_reloc operation returns a valid Dwarf_P_Attribute DIE attribute on success, and DW_DLV_BADADDR if:

- dbg is NULL.
- The Dwarf_P_Debug object contains invalid version information.
- The given ownerdie or otherdie is NULL.
- Attribute does not allow the use of DW_FORM_sec_offset.
- There is not enough memory to allocate internal objects.

dwarf_add_AT_unsigned_LEB128 operation

The dwarf_add_AT_unsigned_LEB128 operation adds an unsigned LEB128 number of form DW_FORM_udata for a given attribute.

Prototype

```
Dwarf_P_Attribute dwarf_add_AT_unsigned_LEB128 (
    Dwarf_P_Die         ownerdie,
    Dwarf_Half          attribute,
    Dwarf_Signed        unsigned_value,
    Dwarf_Error*        error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

ownerdie
Input. This accepts the owning DIE.

attribute
Input. This accepts the DIE attribute.

unsigned_value
Input. This accepts a constant value.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_add_AT_unsigned_LEB128 operation returns the Dwarf_P_Attribute descriptor for attribute on success, and DW_DLV_BADADDR if ownerdie is NULL.

dwarf_add_AT_noninfo_offset operation

The dwarf_add_AT_noninfo_offset operation adds an offset in a section other than .debug_info or .debug_str (that is, DW_FORM_sec_offset).

The offset field is 4 bytes for 32-bit objects, and 8-bytes for 64-bit objects.

Prototype

```
Dwarf_P_Attribute dwarf_add_AT_noninfo_offset (
    Dwarf_P_Debug       dbg,
    Dwarf_P_Die         ownerdie,
    Dwarf_Half          attr,
    Dwarf_Unsigned      offset,
    Dwarf_Error*        error);
```

Parameters

dbg
Input. This accept the Dwarf_P_Debug object.

ownerdie
Input. DIE to receive the given attribute.

attr
Input. DIE attribute name.

offset
Input. Section offset in a section other than .debug_info or .debug_str.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_add_AT_noninfo_offset operation returns a valid Dwarf_P_Attribute DIE attribute on success, and DW_DLV_BADADDR if:

- dbg is NULL.
- The Dwarf_P_Debug object contains invalid version information.
- The given ownerdie is NULL.
- Attribute does not allow the use of DW_FORM_sec_offset.

- There is not enough memory to allocate internal objects.
- There is not enough memory to allocate space to hold offset.

dwarf_die_merge operation

The dwarf_die_merge operation merges the attributes from die_b to die_a.

If the two DIEs are identical, no merge will take place. If usetag_b is true, the tag of die_a will be replaced with the tag of die_b. If usepar_b is true, die_a will inherit the parent of die_b.

Prototype

```
Dwarf_P_Die dwarf_die_merge (
    Dwarf_P_Die         die_a,
    Dwarf_P_Die         die_b,
    Dwarf_Bool          usetag_b,
    Dwarf_Bool          usepar_b,
    Dwarf_Error*        error);
```

Parameters

die_a
Input. The target DIE.

die_b
Input. The source DIE.

usetag_b
Input. Inherit TAG value from source DIE?

usepar_b
Input. Attach target DIE to the parent of the source DIE?

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_die_merge operation returns the target DIE on success, and DW_DLV_BADADDR if dbg is NULL.

Chapter 16. Line-number program (.debug_line) producer operations

These operations create and add information to a line-number program.

dwarf_add_line_entry_b operation

The dwarf_add_line_entry_b operation creates a line-number program and is an alternative method to dwarf_add_line_entry.

dwarf_add_line_entry_b supports compact-flag representation, source view, and sub-line extensions.

Prototype

```
int dwarf_add_line_entry_b(
    Dwarf_P_Debug         dbg,
    Dwarf_Unsigned        file_index,
    Dwarf_Addr            code_address,
    Dwarf_Unsigned        lineno,
    Dwarf_Unsigned        sublineno,
    Dwarf_Signed          column_number,
    Dwarf_Unsigned        view_index,
    Dwarf_Flag            line_std_flags,
    Dwarf_Flag            line_sys_flags,
    Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

file_index
Input. This accepts the index of source-file entries. The entries are from calls to the dwarf_add_file_decl, dwarf_add_lne_file_decl and dwarf_add_global_file_decl APIs.

code_address
Input. This accepts the program address.

lineno
Input. This accepts the source-file line number.

sublineno
Input. This accepts the source-file subline number or 0.

column_number
Input. This accepts the source-file column number or 0.

view_index
Input. This accepts the source-file view index or 0.

line_std_flags
Input. This accepts the standard line-table flags.

line_sys_flags
Input. This accepts the system line-table flags.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_add_line_entry_b operation returns 0 on success and DW_DLV_ERROR if:

- dbg is NULL
- .debug_line section does not exist

dwarf_add_line_entry_b never returns DW_DLV_NO_ENTRY.

dwarf_add_lne_file_decl operation

The dwarf_add_lne_file_decl operation adds a source file declaration.

It results in a DW_LNE_define_file opcode in the body of the current line-number program. dwarf_add_lne_file_decl must be called after all files in the header of the current line-number program have been declared through the dwarf_add_file_decl operation.

Prototype

```
int dwarf_add_lne_file_decl(
    Dwarf_P_Debug         dbg,
    char*                 name,
    Dwarf_Unsigned        dir_index,
    Dwarf_Unsigned        time_last_modified,
    Dwarf_Unsigned        length,
    Dwarf_Unsigned *      ret_src_idx,
    Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

name
Input. This accepts the source-file name.

dir_index
Input. This accepts the source-directory index.

time_last_modified
Input. This accepts the source-file time stamp.

length
Input. This accepts the source-file size.

ret_src_idx
Output. This returns the source-file index.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_add_lne_file_decl operation returns:

- DW_DLV_OK if successful
- DW_DLV_ERROR if:
  - dbg is NULL
  - Return parameter is NULL
  - .debug_line section does not exist

dwarf_add_lne_file_decl never returns DW_DLV_NO_ENTRY.

dwarf_add_global_file_decl operation

The dwarf_add_global_file_decl operation adds a global source-file declaration.

It results in a DW_LNE_IBM_define_global_file opcode in the body of the current line-number program. dwarf_add_global_file_decl must be called after all files in the header of the current line-number program have been declared through the dwarf_add_file_dec operation, and after any files in the body of the current line-number program have been declared through the dwarf_add_lne_file_decl operation.

Prototype

```
int dwarf_add_global_file_decl(
    Dwarf_P_Debug         dbg,
    Dwarf_P_Die           src_die,
    Dwarf_Unsigned *      ret_src_idx,
    Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

src_die
Input. This accepts the source-file DIE object in the .debug_srcfiles section.

ret_src_idx
Output. This returns the source-file index.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_add_global_file_decl operation returns:

- DW_DLV_OK if successful
- DW_DLV_ERROR if:
  - dbg is NULL
  - Return parameter is NULL
  - .debug_line section does not exist

dwarf_add_global_file_decl never returns DW_DLV_NO_ENTRY.

dwarf_line_set_default_isa operation

The dwarf_line_set_default_isa operation sets the default instruction set architecture (ISA).

Prototype

```
int dwarf_line_set_default_isa(
    Dwarf_P_Debug         dbg,
    Dwarf_Unsigned        isa,
    Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

isa
Output. This returns the default ISA value.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_line_set_default_isa operation returns:

- DW_DLV_OK if successful
- DW_DLV_ERROR if dbg is NULL

dwarf_line_set_default_isa never returns DW_DLV_NO_ENTRY.

dwarf_line_set_isa operation operation

The dwarf_line_set_isa operation sets the current instruction set architecture (ISA).

Prototype

```
int dwarf_line_set_isa(
    Dwarf_P_Debug         dbg,
    Dwarf_Unsigned        isa,
    Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

isa
Output. This returns the new ISA value.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_line_set_isa operation returns:

- DW_DLV_OK if successful
- DW_DLV_ERROR if dbg is NULL

dwarf_line_set_isa never returns DW_DLV_NO_ENTRY.

dwarf_global_linetable operation

The dwarf_global_linetable operation switches to global line number table.

All subsequent line-number information is placed in the statement program associated with the CU DIE.

Prototype

```
int dwarf_global_linetable(
    Dwarf_P_Debug         dbg,
    Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_global_linetable operation returns:

- DW_DLV_OK if successful
- DW_DLV_ERROR if:
  - dbg is NULL
  - .debug_info does not exist

dwarf_global_linetable never returns DW_DLV_NO_ENTRY.

dwarf_subprogram_linetable operation

The dwarf_subprogram_linetable operation switches to the subprogram line-number table, which is created on the first call.

All subsequent line-number information is placed in the statement program associated with the subprogram DIE.

Prototype

```
int dwarf_subprogram_linetable(
    Dwarf_P_Debug         dbg,
    Dwarf_P_Die           subpgm_die,
    Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

subpgm_die
Input. This accepts the subprogram DIE object in the .debug_info section.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_subprogram_linetable operation returns:

- DW_DLV_OK if successful
- DW_DLV_ERROR if:
  - dbg is NULL
  - .debug_info does not exist
  - subpgrm_die does not exist

dwarf_subprogram_linetable never returns DW_DLV_NO_ENTRY.

Chapter 17. Location-expression producer APIs

These APIs deal with creation of DWARF location expressions.

dwarf_add_expr_reg operation

The dwarf_add_expr_reg operation takes a given pseudo register and pushes the appropriate DW_OP_reg opcode on the given location expression.

Prototype

```
Dwarf_Unsigned dwarf_add_expr_reg(
    Dwarf_P_Expr          expr,
    Dwarf_Unsigned        reg,
    Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

expr
Input. This accepts the location expression.

reg
Input. This accepts the pseudo register. It must be of the type DW_FRAME_MIPS_REG_type or DW_FRAME_390_REG_type.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_add_expr_reg operation returns the number of bytes in the byte stream for the expr currently generated. It returns DW_DLV_NOCOUNT if:

- expr is NULL
- reg is out of bounds

dwarf_add_expr_breg operation

The dwarf_add_expr_breg operation takes a given pseudo register and a given offset and pushes the appropriate DW_OP_breg opcode on the given location expression.

Prototype

```
Dwarf_Unsigned dwarf_add_expr_breg(
    Dwarf_P_Expr          expr,
    Dwarf_Unsigned        reg,
    Dwarf_Signed          offset,
    Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

expr
Input. This accepts the location expression.

reg
Input. This accepts the pseudo register. It must be of the type DW_FRAME_MIPS_REG_type or DW_FRAME_390_REG_type.

offset
Input. This accepts the offset from the register.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_add_expr_breg operation returns the number of bytes in the byte stream for the expr currently generated. It returns DW_DLV_NOCOUNT if:

- expr is NULL
- reg is out of bounds

dwarf_add_conv_expr operation

The dwarf_add_conv_expr operation pushes a type conversion opcode on the location expression expr. The meaning of val1, val2, and val3 depends on the encoding of the type.

Prototype

```
Dwarf_Unsigned dwarf_add_conv_expr (
    Dwarf_P_Expr        expr,
    Dwarf_Small         opcode,
    Dwarf_Small         f_encoding,
    Dwarf_Unsigned      f_size,
    Dwarf_Small         f_val1,
    Dwarf_Small         f_val2,
    Dwarf_Small         f_val3,
    Dwarf_Small         t_encoding,
    Dwarf_Unsigned      t_size,
    Dwarf_Small         t_val1,
    Dwarf_Small         t_val2,
    Dwarf_Small         t_val3,
    Dwarf_Error         *error);
```

Parameters

expr
Input. This accepts the Dwarf_P_Expr location expression object.

opcode
Input. This accepts a DWARF expression type conversion operator.

f_encoding
Input. This contains the DWARF basetype encoding attribute value for the from operand of the type conversion.

f_size
Input. This contains the size of the from operand of the type conversion in bytes.

f_val1
Input. The first value for describing the from operand of the type conversion.

f_val2
Input. The second value for describing the from operand of the type conversion.

f_val3
Input. The third value for describing the from operand of the type conversion.

t_encoding
Input. This contains the DWARF basetype encoding attribute value for the to operand of the type conversion.

t_size
Input. This contains the size of the to operand of the type conversion in bytes.

t_val1
Input. The first value for describing the to operand of the type conversion.

t_val2
Input. The second value for describing the to operand of the type conversion.

t_val3
Input. The third value for describing the to operand of the type conversion.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_add_conv_expr operation returns the next available byte for pushing operators in the input location expression object. It returns DW_DLV_NOCOUNT if:

- expr is NULL.
- expr does not contain a valid producer debug instance.
- The size of type conversion operands cannot be encoded.
- The opcode value is not supported.
- The total length of the location expression exceeded program limit.

dwarf_add_expr_ref operation

The dwarf_add_expr_ref operation pushes opcode that takes a DIE as operand on the location expression expr.

Prototype

```
Dwarf_Unsigned dwarf_add_expr_ref (
    Dwarf_P_Expr        expr,
    Dwarf_Small         opcode,
    Dwarf_P_Die         die,
    Dwarf_Error         *error);
```

Parameters

expr
Input. This accepts the Dwarf_P_Expr location expression object.

opcode
Input. This accepts a DWARF expression operator that takes a DIE as an operand.

die
Input. The referenced DIE.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_add_expr_ref operation returns the next available byte for pushing operators in the input location expression object. It returns DW_DLV_NOCOUNT if:

- expr is NULL.
- expr does not contain a valid producer debug instance.
- There is not enough memory to allocate internal objects.
- The opcode value is not supported.

dwarf_add_loc_list_entry operation

The dwarf_add_loc_list_entry operation adds a location list entry into the .debug_loc section.

Prototype

```
int dwarf_add_loc_list_entry (
    Dwarf_P_Debug       dbg,
    Dwarf_Addr          begin_addr,
    Dwarf_Addr          end_addr,
    Dwarf_P_Expr        loc_expr,
    Dwarf_Off*          ret_sec_off,
    Dwarf_Error*        error);
```

Parameters

dbg
Input. This accepts the Dwarf_P_Debug object.

begin_addr
Input. The start address to which loc_expr is valid.

end_addr
Input. The end address to which loc_expr becomes invalid.

loc_expr
Input. The location expression that is valid within the given address range.

ret_sec_off
Output. The .debug_loc section offset that points to the beginning of this location list entry. If NULL, this field is not used.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

DW_DLV_OK
The operation is successful. If ret_sec_off is not NULL, it will contain the .debug_loc section offset that points to the beginning of this location list entry.

DW_DLV_NO_ENTRY
Never returned.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- dbg is NULL.
- The Dwarf_P_Debug object contains invalid version information.
- There is not enough memory to allocate internal objects.

dwarf_add_loc_list_base_address_entry operation

The dwarf_add_loc_list_base_address_entry operation adds a base address selection entry into the .debug_loc section.

Prototype

```
int dwarf_add_loc_list_base_address_entry (
    Dwarf_P_Debug       dbg,
    Dwarf_Addr          baseaddr,
    Dwarf_Signed        sym_index,
    Dwarf_Off*          ret_sec_off,
    Dwarf_Error*        error);
```

Parameters

dbg
Input. This accepts the Dwarf_P_Debug object.

baseaddr
Input. A relocatable address which represents the base address for the rest of the location list entries.

sym_index
Input. An ELF symbol table index.

ret_sec_off
Output. The .debug_loc section offset that points to the beginning of this location list entry. If NULL, this field is not used.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

DW_DLV_OK
The operation is successful. If ret_sec_off is not NULL, it will contain the .debug_loc section offset that points to the beginning of this location list entry.

DW_DLV_NO_ENTRY
Never returned.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- dbg is NULL.
- The Dwarf_P_Debug object contains invalid version information.
- There is not enough memory to allocate internal objects.

dwarf_add_loc_list_end_of_list_entry operation

The dwarf_add_loc_list_end_of_list_entry operation adds an end-of-list entry into the .debug_loc section.

Prototype

```
int dwarf_add_loc_list_end_of_list_entry (
    Dwarf_P_Debug       dbg,
    Dwarf_Error*        error);
```

Parameters

dbg
Input. This accepts the Dwarf_P_Debug object.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

DW_DLV_OK
The operation is successful. An end-of-list entry is added into the .debug_loc section.

DW_DLV_NO_ENTRY
Never returned.

DW_DLV_ERROR
Returned if either of the following conditions apply:

- dbg is NULL.

- The Dwarf_P_Debug object contains invalid version information.
- There is not enough memory to allocate internal objects.

Chapter 18. Accelerated access producer operation

The APIs in this section create entries in a fast-access debug section.

dwarf_add_pubtype operation

The dwarf_add_pubtype operation defines a global type name in .debug_pubtypes.

Prototype

```
Dwarf_Unsigned dwarf_add_pubtype(
    Dwarf_P_Debug         dbg,
    Dwarf_P_Die           die,
    char*                 pubtype_name,
    Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

die
Input. This accepts a file-scoped user defined type DIE.

pubtype_name
Input. This accepts the name of the public type.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_add_pubtype operation returns a non-zero value on success, and returns zero if:

- dbg is NULL.
- die is NULL.
- pubtype_name is NULL.

Chapter 19. Dynamic storage management operation

The operation in this section controls the dynamic storage within the libdwarf producer object.

dwarf_p_dealloc

The dwarf_p_dealloc API frees the dynamic storage pointed to by a given space address and allocated to the given Dwarf_P_Debug.

Prototype

```
void dwarf_p_dealloc(
    Dwarf_P_Debug         dbg,
    Dwarf_Ptr             space,
    Dwarf_Unsigned        type);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

space
Input. This accepts the storage address.

type
Input. This accepts the storage allocation type.

Return values

The dwarf_p_dealloc API does not have a return value.

Chapter 20. Range-list producer APIs

Range-list producer operations update the .debug_ranges section.

dwarf_add_range_list_entry operation

The dwarf_add_range_list_entry operation adds a range-list entry.

The addresses are either offset from DW_AT_low_pc of the CU, or based on a specified address-selection entry.

Prototype

```
int dwarf_add_range_list_entry (
  Dwarf_P_Debug         dbg,
  Dwarf_Addr            begin_addr,
  Dwarf_Addr            end_addr,
  Dwarf_Off*            ret_sec_off,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

begin_addr
Input. This accepts the starting address.

end_addr
Input. This accepts the final address.

ret_sec_off
Output. This returns the section offset in the .debug_ranges section. This can be NULL, if the section is not needed.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_add_range_list_entry operation returns:

- DW_DLV_OK if successful
- DW_DLV_ERROR if dbg is NULL

dwarf_add_range_list_entry never returns DW_DLV_NO_ENTRY.

dwarf_add_base_address_entry operation

The dwarf_add_base_address_entry operation adds a base address-selection entry.

Prototype

```
int dwarf_add_base_address_entry (
  Dwarf_P_Debug         dbg,
  Dwarf_Addr            baseaddr,
  Dwarf_Off*            ret_sec_off,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

baseaddr
Input. This accepts the starting address.

ret_sec_off
Output. This returns the section offset in the .debug_ranges section.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_add_base_address_entry operation returns:

- DW_DLV_OK if successful
- DW_DLV_ERROR if dbg is NULL

dwarf_add_base_address_entry never returns DW_DLV_NO_ENTRY.

dwarf_add_end_of_list_entry operation

The dwarf_add_end_of_list_entry operation adds an end-of-list entry.

Prototype

```
int dwarf_add_end_of_list_entry  (
  Dwarf_P_Debug         dbg,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

The dwarf_add_end_of_list_entry operation returns:

- DW_DLV_OK if successful
- DW_DLV_ERROR if dbg is NULL

dwarf_add_end_of_list_entry never returns DW_DLV_NO_ENTRY.

Chapter 21. Producer flag operations

These operations query and set the flags that are used by the producer operations.

dwarf_pro_flag_any_set operation

The dwarf_pro_flag_any_set operation tests whether or not any of the Dwarf_Flag index bit are set.

Prototype

```
int dwarf_pro_flag_any_set (
  Dwarf_P_Debug         dbg,
  Dwarf_Flag*           flags,
  Dwarf_Bool*           ret_anyset,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

flags
Input/Output. This accepts or returns a Dwarf_Flag object.

ret_anyset
Output. This returns the Boolean value which indicates whether or not any bit index is set.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

dwarf_pro_flag_any_set returns DW_DLV_ERROR if the returned parameter is NULL and it never returns DW_DLV_NO_ENTRY.

Memory allocation

There is no storage to deallocate.

dwarf_pro_flag_clear operation

The dwarf_pro_flag_clear operation clears the given Dwarf_Flag index bit.

Prototype

```
int dwarf_pro_flag_clear (
  Dwarf_P_Debug         dbg,
  Dwarf_Flag*           flags,
  int                   bit_idx,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

flags
Input/Output. This accepts or returns a Dwarf_Flag object.

bit_idx
Input. This accepts the flag bit index to clear. It can be a value from 0 to 31.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

dwarf_pro_flag_clear returns DW_DLV_ERROR if the returned parameter is NULL and it never returns DW_DLV_NO_ENTRY.

Memory allocation

There is no storage to deallocate.

dwarf_pro_flag_complement operation

The dwarf_pro_flag_complement operation complements the given Dwarf_Flag index bit.

Prototype

```
int dwarf_pro_flag_complement (
  Dwarf_P_Debug         dbg,
  Dwarf_Flag*           flags,
  int                   bit_idx,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

flags
Input/Output. This accepts or returns a Dwarf_Flag object.

bit_idx
Input. This accepts the flag bit index to complement. It can be a value from 0 to 31.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

dwarf_pro_flag_complement returns DW_DLV_ERROR if the returned parameter is NULL and it never returns DW_DLV_NO_ENTRY.

Memory allocation

There is no storage to deallocate.

dwarf_pro_flag_copy operation

The dwarf_pro_flag_copy operation sets or clears the given Dwarf_Flag index bit.

The action is determined by a given Boolean value.

Prototype

```
int dwarf_pro_flag_copy (
  Dwarf_P_Debug         dbg,
  Dwarf_Flag*           flags,
  int                   bit_idx,
```

```
  Dwarf_Bool            val,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

flags
Input/Output. This accepts or returns a Dwarf_Flag object.

bit_idx
Input. This accepts the flag bit index to set or clear. It can be a value from 0 to 31.

val
Input. This accepts the Boolean value which indicates whether to set or clear the bit index.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

dwarf_pro_flag_copy returns DW_DLV_ERROR if the returned parameter is NULL and it never returns DW_DLV_NO_ENTRY.

Memory allocation

There is no storage to deallocate.

dwarf_pro_flag_reset operation

The dwarf_pro_flag_reset operation clears all the Dwarf_Flag index bits of a given libdwarf consumer object.

Prototype

```
int dwarf_pro_flag_reset (
  Dwarf_P_Debug         dbg,
  Dwarf_Flag*           flags,
  Dwarf_Error*          error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

flags
Input/Output. This accepts or returns a Dwarf_Flag object.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

dwarf_pro_flag_reset returns DW_DLV_ERROR if the returned parameter is NULL and it never returns DW_DLV_NO_ENTRY.

Memory allocation

There is no storage to deallocate.

dwarf_pro_flag_set operation

The dwarf_pro_flag_set operation sets the given Dwarf_Flag index bit.

Prototype

```
int dwarf_pro_flag_set (
  Dwarf_P_Debug          dbg,
  Dwarf_Flag*            flags,
  int                    bit_idx,
  Dwarf_Error*           error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

flags
Input/Output. This accepts or returns a Dwarf_Flag object.

bit_idx
Input. This accepts the flag bit index to set. It can be a value from 0 to 31.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

dwarf_pro_flag_set returns DW_DLV_ERROR if the returned parameter is NULL and it never returns DW_DLV_NO_ENTRY.

Memory allocation

There is no storage to deallocate.

dwarf_pro_flag_test operation

The dwarf_pro_flag_test operation tests whether or not the given Dwarf_Flag index bit is set.

Prototype

```
int dwarf_pro_flag_test (
  Dwarf_P_Debug          dbg,
  Dwarf_Flag*            flags,
  int                    bit_idx,
  Dwarf_Bool*            ret_bitset,
  Dwarf_Error*           error);
```

Parameters

dbg
Input. This accepts a libdwarf producer object.

flags
Input/Output. This accepts or returns a Dwarf_Flag object.

bit_idx
Input. This accepts the flag bit index to test. It can be a value from 0 to 31.

ret_bitset
Output. This returns the Boolean value which indicates whether or not the bit index is set.

error
Input/output. This accepts or returns the Dwarf_Error object.

Return values

`dwarf_pro_flag_test` returns `DW_DLV_ERROR` if the returned parameter is NULL and it never returns `DW_DLV_NO_ENTRY`.

Memory allocation

There is no storage to deallocate.

Chapter 22. IBM extensions to libelf

IBM extensions to the `libelf` library facilitate the creation of ELF objects for different platforms and file systems. ELF objects are used to store the DWARF debugging information.

Extensions to the `libelf` library are categorized as follows:

ELF initialization and termination APIs

ELF initialization and termination APIs are IBM extensions to the `libelf` library that facilitate the creation of ELF objects for different platforms and file systems. ELF objects are used to store the DWARF debugging information.

Elf_Alloc_Func object

If an Elf_Mem_Image object is used to create the ELF object file, the Elf operation will use the user-specified memory deallocation function to get storage used for the ELF object file.

Type definition

```
typedef void* (*Elf_Alloc_Func) (size_t size);
```

Elf_Dealloc_Func object

If an `Elf_Mem_Image` object is used to create the ELF object file, the Elf operation will use the user-specified memory allocation function to free storage for the ELF object file.

Type definition

```
typedef void (*Elf_Dealloc_Func) (void* p);
```

Elf_Mem_Image object

An opaque datatype for accessing an ELF object file that is stored in memory.

Type definition

```
typedef struct Elf_Mem_Image_s* Elf_Mem_Image;
```

elf_begin_b operation

The `elf_begin_b` operation is used to read from and write to an ELF descriptor.

`elf_begin_b` is similar to `elf_begin` except that it accesses the ELF descriptor with a file pointer returned from the `fopen` function.

Prototype

```
Elf * elf_begin_b (
    FILE *    __fp,
    Elf_Cmd   __cmd,
    Elf *     __ref);
```

Parameters

__fp
Input. This accepts a file pointer to the ELF descriptor. The pointer is returned from the `fopen` function.

__cmd
Input. This accepts the ELF access mode.

__ref
Input. This accepts the return from the previous `elf_begin`, `elf_begin_b`, or `elf_begin_c` API.

Memory allocation

`elf_end` is used to terminate the ELF descriptor and deallocate the memory associated with the descriptor.

elf_begin_c operation

The `elf_begin_c` operation is used to initialize and obtain an ELF descriptor. `elf_begin_c` might read an existing file, update an existing file, or create a new file. Before the first call to the `elf_begin_c` operation, a program must call the `elf_version` operation to coordinate versions.

Prototype

```
Elf * elf_begin_c (
    ELF_Mem_Image        elf_mem_image,
    Elf_Cmd                  cmd,
    Elf *                    ref);
```

Parameters

elf_mem_image
Input. Contains a memory image of the ELF object file .

cmd
Input. This specifies the command that obtains the ELF access mode.

- The ELF_C_NULL command returns a NULL pointer, without opening a new descriptor.
- The ELF_C_READ command examines the contents of the memory image. The API allocates a new ELF descriptor and prepares to process the entire ELF object file.
- The ELF_C_RDWR command duplicates the actions of ELF_C_READ and then allows the API to update the memory image.

Note: The ELF_C_READ command gives a read-only view of the file, while the ELF_C_RDWR command lets the API read and write the file.

ref
Input. Intended for supporting archive files. Currently not supported on z/OS. User must specify NULL as input.

Return values

Returns NULL if ELF_C_NULL is specified as the command, or an error has occurred. Otherwise, returns a non-NULL ELF descriptor.

Cleanups

The `elf_end` operation is used to terminate the ELF descriptor and deallocate the memory associated with the descriptor, as shown in Figure 1 on page 193.

```
Elf* elf;
Elf_Mem_Image image;

// Coordinate ELF version
elf_version (EV_CURRENT);

// The ELF object is 1000 bytes long, and is stored in 'buffer'
image = elf_create_mem_image (buffer, 1000, NULL, NULL);

// Examine ELF object for reading
elf = elf_begin_c (image, ELF_C_READ, NULL);

// terminate 'elf' (optional)
elf_end(elf);

// terminate Elf_Mem_Image
elf_term_mem_image (image);
```

Figure 1. Example: Code that terminates an ELF descriptor and deallocates memory

elf_create_mem_image operation

If the ELF object is stored in memory (not in physical file), use this operation to create an Elf_Mem_Image object for reading or writing.

Prototype

```
Elf_Mem_Image
  elf_create_mem_image(
    char*            buf,
    long             length,
    Elf_Alloc_Func   alloc_func,
    Elf_Dealloc_Func dealloc_func);
```

Parameters

buf
Input. Memory pointer to the start of the ELF object. Specify NULL if the purpose is to create a new ELF object in memory.

length
Input. Length of the ELF object. This field is ignored if the purpose is to create a new ELF object in memory.

alloc_func
Input. Elf operations use this memory allocation function to get storage during creation of the ELF object file. This field is ignored if the purpose is to read an ELF object.

dealloc_func
Input. Elf operations use this memory deallocation function to free storage during creation of the ELF object file. This field is ignored if the purpose is to read an ELF object.

Return values

Returns NULL if there is not enough memory to allocate the Elf_Mem_Image object. Otherwise, returns an initialized Elf_Mem_Image object.

Cleanups

elf_term_mem_image is used to terminate the Elf_Mem_Image object and deallocate the memory associated with the descriptor.

Example

```
Elf*           elf;
Elf_Mem_Image  image;

// Coordinate ELF version
elf_version (EV_CURRENT);
```

```
// Create an Elf_Mem_Image in memory to store ELF object
image = elf_create_mem_image (NULL, 0, malloc, free);

// Create ELF object for writing
elf = elf_begin_c (image, ELF_C_WRITE, NULL);

// terminate 'elf' (optional)
elf_end(elf);
// terminate Elf_Mem_Image
elf_term_mem_image (image);
```

elf_get_mem_image operation

This operation retrieves the memory image from the Elf_Mem_Image object.

Prototype

```
int
  elf_get_mem_image(
    Elf_Mem_Image    elf_mem_image,
    char**           buf,
    long*            length);
```

Parameters

elf_mem_image
Input. Accepts the Elf_Mem_Image object containing the ELF object.

buf
Output. Returns a pointer to the ELF object held in memory

length
Output. ReturnS the length of the ELF object held in memory.

Return values

Returns 1 if the returned parameters are NULL, or if the Elf_Mem_Image object is NULL. Otherwise, this returns 0.

Cleanups

None.

elf_term_mem_image operation

This operation terminates the Elf_Mem_Image object and deallocates the memory associated with the descriptor.

Prototype

```
void
  elf_term_mem_image(
    Elf_Mem_Image    elf_mem_image);
```

Parameters

elf_mem_image
Input. The input Elf_Mem_Image object containing the ELF object

Return values

None.

Cleanups

None.

ELF utilities

ELF utilities manipulate ELF executable objects.

elf_build_version operation

This operation displays the build ID of the elf library. Every release/PTF of the elf library will have an unique build ID. This information is useful for providing service information to IBM customer support. Calling this function will emit the build ID string (encoded in ISO8859-1) to stdout.

BLD_LEVEL is an unsigned integer. `elf_build_version` can then query this build-level value.

Prototype

```
char*
  elf_build_version (void);
```

Return values

`elf_build_version` only returns the build ID of the elf library. The returned string is encoded in ISO8859-1.

Example

```
/* Compile this code with ASCII option */
printf ("Library(elf) Level(%s)\n", elf_build_version());
```

elf_dll_version operation

This operation validates the version of the DLL, and should be used when dynamically linking to the `libelf` or `libdwarf` library. To retrieve the current library version, call the function with '-1' as an argument.

If the call is successful, '0' is returned. Otherwise, the version value LIBELF_DLL_VERSION is returned inside the DLL.

Prototype

```
unsigned int
   elf_dll_version(
     unsigned int         ver);
```

Parameters

ver
Version of current DLL, which can be obtained using the LIBELF_DLL_VERSION macro found in `libelf.h`.

Return values

0
The DLL version is compatible. The user code is compiled with an elf/dwarf DLL that is the same as the current one, or perhaps earlier.

Any non-zero value
The version of the elf/dwarf DLL used for building the user code, means that the user code is compiled with an elf/dwarf DLL that is more recent than the current library and is incompatible.

Example

```
#include
#include "libelf.h"

dllhandle     *cdadll;
unsigned int (*version_chk)(unsigned int);
unsigned int  dll_version;

#ifdef _LP64
 #define __CDA_ELF   "CDAEQED"
#else
 #define __CDA_ELF   "CDAEED"
#endif

#if LIBELF_IS_DLL
 cdadll = dllload(__CDA_ELF);
 if (cdadll == NULL) {
   /* elf/dwarf DLL not found */
 }

 version_chk = (unsigned int (*)(unsigned int))
           dllqueryfn(cdadll, "elf_dll_version");
 if (version_chk == NULL) {
   /* Version API not found, should NEVER happen */
 }

 dll_version = version_chk (LIBELF_DLL_VERSION);
 if (dll_version != 0) {
   /* Incompatible DLL version */
 }
#endif
```

Appendix A. Diagnosing Problems

The following information describes how to determine the source of errors in your code.

Limitation of service

Service is limited to IBM customers through the normal service channels.

Diagnosis checklist

This checklist is designed to either solve your problem or help you gather the diagnostic information required for determining the source of the error. It can help you confirm if the suspected failure is a user error caused by incorrect usage of the `libelf` or `libdwarf` library or by an error in the logic of the routine.

Step through each of the items in the diagnosis checklist below to see if they apply to your problem:

1. If your failing application contains programs that were changed since they last ran successfully, review the output of the compile or assembly (listings) for any unresolved errors.
2. If you are an IBM customer, your installation may have received an IBM Program Temporary Fix (PTF) for the problem. Verify that you have received all issued PTFs and have installed them, so that your installation is at the most current maintenance level.
3. If you are an IBM customer, the preventive service planning (PSP) bucket, an online database available through IBM service channels, gives information about product installation problems and other problems. Check to see whether it contains information related to your problem.
4. Narrow the source of the error:
   - Verify that either the `libdwarf` or `libelf` DLL exists. You can use the following code to see if the DLL can be found during execution.

```
#define _UNIX03_SOURCE
#include <dlfcn.h> /* dlopen,dlsym,dlclose */
#include "libelf.h"

void *cdadll;
unsigned int (*version_chk)(unsigned int);
unsigned int dll_version;

#ifdef _LP64
#define __CDA_ELF "CDAEQED"
#else
#define __CDA_ELF "CDAEED"
#endif

#if LIBELF_IS_DLL
cdadll = dlopen(__CDA_ELF, RTLD_LOCAL | RTLD_LAZY);
if (cdadll == NULL) {
/* elf/dwarf DLL not found */
}

version_chk = (unsigned int (*)(unsigned int))
dlsym(cdadll, "elf_dll_version");
if (version_chk == NULL) {
/* Version API not found, should NEVER happen */
}

dll_version = version_chk (LIBELF_DLL_VERSION);
if (dll_version != 0) {
/* Incompatible DLL version */
}
dlclose(cdadll);
#endif
</dlfcn.h>
```

- Verify that either the `libdwarf` or `libdelf` version is correct. You can use the following code to verify the version:

```
if (elf_dll_version(LIBELF_DLL_VERSION) != 0) {
  /* Version mismatched */
  /* Make sure your application is compiled with the
     libdwarf/libelf header file that are found together
     with the DLL module */
}
```

- Verify that an abend is caused by product failures and not by program errors. By reading the CEEDUMP, you can identify if the abends happens within either the `libdwarf` or `libdelf` module. Figure 2 on page 198 shows that the `dwarf_producer_init_b` API (highlighted in bold letters) is causing the abend:

5. After you identify the failure, consider writing a small test case that recreates the problem. The test case could help you determine if the error is in a user routine or in either the `libdwarf` or `libdelf` library. Do not make the test case larger than 75 lines of code. The test case is not required, but it could expedite the process of finding the problem.

   If the error is not a `libdwarf` or `libdelf` library failure, refer to the diagnosis procedures for the product that failed.
6. Record the conditions and options in effect at the time the problem occurred. Compile your program with the appropriate options to obtain an assembler listing and data map. If possible, obtain the binder or linkage-editor output listing. Note any changes from the previous successful compilation or run. For an explanation of compiler options, refer to the compiler-specific programming guide.
7. If you are experiencing a no-response problem, try to force a dump, and cancel the program with the dump option.
8. Record the sequence of events that led to the error condition and any related programs or files. It is also helpful to record the service-level of the compiler associated with the failing program.

```
CEE3DMP V1 R5.0: Condition processing resulted in the unhandled condition.
Information for enclave main

  Information for thread 282D51C000000000

  Traceback:
    DSA Addr  Program Unit  PU Addr   PU Offset  Entry         E Addr    E  Offset   Statement  Load Mod  Service  Status
    2867EF08  CEEHDSP       281A1068  +00004808  CEEHDSP       281A1068  +00004808              CEEPLPKA  DRIVER5  Call
    2867E350  CEEHRNUH      281ADD60  +00000086  CEEHRNUH      281ADD60  +00000086              CEEPLPKA  DRIVER5  Call
    28731560                28845FE8  +00000080  dwarf_producer_init_b
                                                                28845FE8  +00000080         202  CDAEED             Exception
                            27E1F070  +00000678  main          27E1F070  +00000678         360  *PATHNAM           Call
    28731720                28275398  +000009AA  CEEVROND      282753F0  +00000952              CEEPLPKA           Call
    2867E0F8  EDCZHINV      285E8250  +0000009A  EDCZHINV      285E8250  +0000009A              CELHV003  DRIVER5  Call
    2867E030  CEEBBEXT      28173B70  +000001A6  CEEBBEXT      28173B70  +000001A6              CEEPLPKA  DRIVER5  Call
```

Figure 2. Example of traceback of condition processing that resulted in an unhandled condition

Appendix B. Accessibility

Accessible publications for this product are offered through IBM Documentation (www.ibm.com/docs/en/zos).

If you experience difficulty with the accessibility of any z/OS information, send a detailed message to the Contact the z/OS team web page (www.ibm.com/systems/campaignmail/z/zos/contact_z) or use the following mailing address.

IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
United States

Accessibility features

Accessibility features help users who have physical disabilities such as restricted mobility or limited vision use software products successfully. The accessibility features in z/OS can help users do the following tasks:

- Run assistive technology such as screen readers and screen magnifier software.
- Operate specific or equivalent features by using the keyboard.
- Customize display attributes such as color, contrast, and font size.

Consult assistive technologies

Assistive technology products such as screen readers function with the user interfaces found in z/OS. Consult the product information for the specific assistive technology product that is used to access z/OS interfaces.

Keyboard navigation of the user interface

You can access z/OS user interfaces with TSO/E or ISPF. The following information describes how to use TSO/E and ISPF, including the use of keyboard shortcuts and function keys (PF keys). Each guide includes the default settings for the PF keys.

- *z/OS TSO/E Primer*
- *z/OS TSO/E User's Guide*
- *z/OS ISPF User's Guide Vol I*

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users who access IBM Documentation with a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line because they are considered a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that the screen reader is set to read out punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1)

are mutually exclusive alternatives. If you hear the lines `3.1 USERID` and `3.1 SYSTEMID`, your syntax can include either `USERID` or `SYSTEMID`, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The `*` symbol is placed next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element `*FILE` with dotted decimal number 3 is given the format `3 \* FILE`. Format `3* FILE` indicates that syntax element `FILE` repeats. Format `3* \* FILE` indicates that syntax element `* FILE` repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol to provide information about the syntax elements. For example, the lines `5.1*, 5.1 LASTRUN, and 5.1 DELETE` mean that if you use more than one of the `LASTRUN` and `DELETE` syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, it indicates a reference that is defined elsewhere. The string that follows the % symbol is the name of a syntax fragment rather than a literal. For example, the line `2.1 %OP1` means that you must refer to separate syntax fragment `OP1`.

The following symbols are used next to the dotted decimal numbers.

? indicates an optional syntax element
The question mark (?) symbol indicates an optional syntax element. A dotted decimal number followed by the question mark symbol (?) indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example `5? NOTIFY`). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines `5 ?, 5 NOTIFY`, and `5 UPDATE`, you know that the syntax elements `NOTIFY` and `UPDATE` are optional. That is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.

! indicates a default syntax element
The exclamation mark (!) symbol indicates a default syntax element. A dotted decimal number followed by the `!` symbol and a syntax element indicate that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the dotted decimal number can specify the ! symbol. For example, if you hear the lines `2? FILE, 2.1! (KEEP), and 2.1 (DELETE)`, you know that `(KEEP)` is the default option for the `FILE` keyword. In the example, if you include the `FILE` keyword, but do not specify an option, the default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the `FILE` keyword is omitted, the default `FILE(KEEP)` is used. However, if you hear the lines `2? FILE, 2.1, 2.1.1! (KEEP)`, and `2.1.1 (DELETE)`, the default option KEEP applies only to the next higher dotted decimal number, `2.1` (which does not have an associated keyword), and does not apply to `2? FILE`. Nothing is used if the keyword `FILE` is omitted.

*** indicates an optional syntax element that is repeatable**
The asterisk or glyph (*) symbol indicates a syntax element that can be repeated zero or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line `5.1* data area`, you know that you can include one data area, more than one data area, or no data area. If you hear the lines `3* , 3 HOST, 3 STATE`, you know that you can include `HOST, STATE`, both together, or nothing.

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you can write `HOST STATE`, but you cannot write `HOST HOST`.
3. The * symbol is equivalent to a loopback line in a railroad syntax diagram.

+ indicates a syntax element that must be included

The plus (+) symbol indicates a syntax element that must be included at least once. A dotted decimal number followed by the + symbol indicates that the syntax element must be included one or more times. That is, it must be included at least once and can be repeated. For example, if you hear the line `6.1+ data area`, you must include at least one data area. If you hear the lines `2+, 2 HOST,` and `2 STATE`, you know that you must include `HOST, STATE,` or both. Similar to the * symbol, the + symbol can repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loopback line in a railroad syntax diagram.

Notices

This information was developed for products and services that are offered in the USA or elsewhere.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
United States of America

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

This information could include missing, incorrect, or broken hyperlinks. Hyperlinks are maintained in only the HTML plug-in output for IBM Documentation. Use of hyperlinks in other output formats of this information is at your own risk.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Site Counsel
2455 South Road

Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or

reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Online Privacy Statement

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

Depending upon the configurations deployed, this Software Offering may use session cookies that collect each user's name, email address, phone number, or other personally identifiable information for purposes of enhanced user usability and single sign-on configuration. These cookies can be disabled, but disabling them will also eliminate the functionality they enable.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at ibm.com®/privacy and IBM's Online Privacy Statement at ibm.com/privacy/details in the section entitled "Cookies, Web Beacons and Other Technologies," and the "IBM Software Products and Software-as-a-Service Privacy Statement" at ibm.com/software/info/product-privacy.

Policy for unsupported hardware

Various z/OS elements, such as DFSMSdfp, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those

products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: IBM Lifecycle Support for z/OS (www.ibm.com/software/support/systemsz/lifecycle)
- For information about currently-supported IBM hardware, contact your IBM representative.

Programming interface information

This publication documents intended Programming Interfaces that allow the customer to write programs to obtain services of Common Debug Architecture.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and Trademark information (www.ibm.com/legal/copytrade.shtml).

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Standards

The libddpi library supports the DWARF Version 3 and Version 4 format and ELF application binary interface (ABI).

DWARF was developed by the UNIX International Programming Languages Special Interest Group (SIG). CDA's implementation of DWARF is based on the DWARF 4 standard.

ELF was developed as part of the System V ABI. It is copyrighted 1997, 2001, The Santa Cruz Operation, Inc. All rights reserved.

Index

Product Number: 5650-ZOS

SC14-7312-50