z/OS

IBM

DWARF/ELF Extensions Library Reference

SC09-7655-03

z/OS

DWARF/ELF Extensions Library Reference

SC09-7655-03

Note:
Before using this information and the product it supports, read the information in "Notices" on page 143.

Fourth Edition (September 2010)

This edition applies to Run-Time Library Extensions in Version 1 Release 12 of z/OS (5694-A01) and to all subsequent releases until otherwise indicated in new editions. This edition replaces SC09-7655-02. Ensure that you apply all necessary PTFs for the program.

IBM welcomes your comments. You can send your comments via e-mail to compinfo@ca.ibm.com. Be sure to include your e-mail address if you want a reply.

Include the title and order number of this document, and the page number or topic related to your comment. 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.

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:

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" on page 1 and "Example of a DWARF file" on page 2.

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 XL C/C++ compiler 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 on z/OS

- Writing debugging programs in XL C/C++ 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, and ELF publications

Document title and number	Key sections/chapters in the document
z/OS Common Debug Architecture Library Reference, SC09-7654	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. See http://www.ibm.com/software/awdtools/libraryext/library/.
z/OS Common Debug Architecture User's Guide, SC09-7653	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. See http://www.ibm.com/software/awdtools/libraryext/library/.
System V Application Binary Interface Standard	The Draft April 24, 2001 version of the ELF standard. For more information, go to: http://www.ibm.com/software/awdtools/libraryext/library/.
ELF Application Binary Interface Supplement	The Draft April 24, 2001 version of the ELF standard supplement. For more information, go to: http://www.ibm.com/software/awdtools/libraryext/library/.
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. See http://www.ibm.com/software/awdtools/libraryext/library/.
Producer Library Interface to DWARF	The revision 1.18, January 10, 2002, version of the `libdwarf` producer library. See http://www.ibm.com/software/awdtools/libraryext/library/.
MIPS Extensions to DWARF Version 2.0	The revision 1.17, August 29, 2001, version of the MIPS extension to DWARF. See http://www.ibm.com/software/awdtools/libraryext/library/.

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

Document title and number	Key sections/chapters in the document
z/OS XL C/C++ User's Guide, SC09-4767	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, as, CDAHLASM) • Diagnosing problems • Cataloged procedures and REXX EXECs supplied by IBM See http://www.ibm.com/software/awdtools/czos/library.
z/OS XL C/C++ Programming Guide, SC09-4767	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

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*
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 formats and IBM BookMaster® formats on the following CD: *z/OS Collection, SK3T-4269*. They are also available at the following Web site: www.ibm.com/software/awdtools/libraryext/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 www.adobe.com.

You can also browse the documents on the World Wide Web by visiting the z/OS library at www.ibm.com/servers/eserver/zseries/zos/bkserv/.

Note: For further information on viewing and printing softcopy documents and using IBM BookManager®, see *z/OS Information Roadmap*.

Where to find more information

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

Run-Time Library Extensions on the World Wide Web

Additional information on Common Debug Architecture is available on the World Wide Web on the Run-Time Library Extensions home page at: http://www.ibm.com/software/awdtools/libraryext/

This page contains links to other useful information, including the Run-Time Library Extensions information library, which includes the Common Debug Architecture documents.

Information updates on the web

For the latest information updates that have been provided in PTF cover letters and Documentation APARs for IBM z/OS, refer to the online list of APARs and PTFs. This document is updated weekly and lists documentation changes before they are incorporated into z/OS publications.

The online list of APARs and PTFs is found at: http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/BOOKS/ZIDOCMST/CCONTENTS

Technical support

Additional technical support is available from the z/OS XL C/C++ Support page. This page provides a portal with search capabilities to a large selection of technical support FAQs and other support documents.

You can find the z/OS XL C/C++ Support page on the Web at: www.ibm.com/software/awdtools/czos/support.

Readme files that include changes to z/OS XL C/C++ publications are available from: www.ibm.com/support/docview.wss?uid=swg27007531.

If you cannot find what you need, you can e-mail:

compinfo@ca.ibm.com

For the latest information about z/OS XL C/C++, visit the product information site at: www.ibm.com/software/awdtools/czos/.

For information about boosting performance, productivity and portability, visit the C/C++ Cafe at: www.ibm.com/software/rational/cafe/community/ccpp.

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 z/OS XL C/C++ documentation, send your comments by e-mail to: compinfo@ca.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 store DWARF debugging information in separate ELF object files, rather than a typical object file. When the executable module is loaded into memory.

- The size of the executable module is reduced.
- Memory usage is minimized.

The DWARF industry-standard debugging information format

The DWARF 3 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. You use the DEBUG option of the z/OS XL C/C++ compiler 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.

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 DIE index number (from 0 to n-1) that uniquely identifies each DIE. The index number is not found in the DWARF file.
  - 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. This avoids the need to relocate the debugging data, which speeds up program loading and debugging.
  - A list of attributes, which fills in details and further describes the entity. Each attribute name has the DW_ATT 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 http://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 parent 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 DWARF that are used to create 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 compiled with the XPLINK and ASCII options. For more information about these compiler options, refer to *z/OS XL C/C++ User's Guide*.

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
- z/OS future support for languages such as FORTRAN, HLASM, COBOL, PL/I, PL/X, PL/IX, and Pascal

Changes to DWARF/ELF library extensions for z/OS V1R12

CDA libraries shipped with z/OS V1R12 include additional attribute to the `DW_TAG_IBM_src_file` tag.

See "Source-file entries" on page 49 for information about the following attribute:
- `DW_AT_IBM_charset`

There are also updates to "DW_TAG_src_file DIE tag" on page 49.

Chapter 2. 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 handling macros

The following are error handling macros:

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_VIEW_ENTRY_ALLOC
Value = 199. Memory allocation failed in creating global source-view entry.

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

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_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 has invalid 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 NULL.

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 NULL.

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

DW_DLE_DEBUG_SRCINFO_NULL
Value = 221. The .debug_srcinfo section is NULL.

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 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 NULL.

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 relocation section for .debug_info is NULL.

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 relocation section for .debug_line is NULL.

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_INVALID_VIEW
Value = 250. The source view was not found.

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 invalid.

DW_DLE_CODESET_CONVERSION_ERROR
Value = 253. Error converting between codesets.

DW_DLE_STRING_NULL
Value = 254. Dwarf string object is NULL.

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

These operations are related to creating and terminating `libdwarf` consumer 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_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(...).

ELF symbol table and section consumer operations

Section consumer operations query information from the ELF symbol table (`.symtab` section) within the ELF file.

Debug sections

Example of a typical ELF symbol table.

```
Section 16 .symtab
    Sym  0: value= 0x000, size=    0 sect= undef,           type= none, bind= local
    Sym  1: value= 0x000, size= 1056 sect= .text,           type= sect, bind= local
    Sym  2: value= 0x408, size=    0 sect= .text,           type= none, bind= local
    Sym  3: value= 0x000, size=    1 sect= .debug_info,     type= sect, bind= local
    Sym  4: value= 0x000, size=    1 sect= .debug_line,     type= sect, bind= local
    Sym  5: value= 0x000, size=    1 sect= .debug_abbrev,   type= sect, bind= local
    Sym  6: value= 0x000, size=    1 sect= .debug_aranges,  type= sect, bind= local
    Sym  7: value= 0x000, size=    1 sect= .debug_pubnames, type= sect, bind= local
    Sym  8: value= 0x000, size=    1 sect= .debug_ppa,      type= sect, bind= local
    Sym  9: value= 0x000, size=    1 sect= .debug_srcfiles, type= sect, bind= local
```

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.

Debugging-section consumer APIs

Standard DWARF operations assume that all debug queries are from the `.debug_info` section, which is the only DIE section. IBM extensions provide additional DIE sections and operations that will work on all sections. The consumers for the `.debug_info` section use two types: `Dwarf_section_type` and `Dwarf_section_content`.

Extensions to DWARF debugging sections

IBM extensions to DWARF debugging sections.

The extended sections are:

- .debug_info
- .debug_srcfiles
- .debug_ppa

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 ELF symbol table.

See “Debug sections” on page 12.

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_AUTOMT      =  18,
  DW_SECTION_NUM_SECTIONS
} Dwarf_section_type;
```

In this case, the only valid types are:

```
DW_SECTION_DEBUG_INFO = 0,
DW_SECTION_DEBUG_PPA = 15,
DW_SECTION_DEBUG_SRCFILES = 16,
```

Note: DW_SECTION_NUM_SECTIONS must be the last entry in this structure.

Members

Currently, only the following members are supported for the consumer debugging-section operations:

DW_SECTION_DEBUG_INFO
Contains `.debug_info` information.

DW_SECTION_DEBUG_PPA
Contains `.debug_ppa` information.

DW_SECTION_DEBUG_SRCFILES
Contains `.debug_srcfiles` information.

Dwarf_section_content object

The `Dwarf_section_content` data structure is used to show differences in the contents of the `Dwarf_section_type` section.

Type definition

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

Note: DW_SECTION_NUM_SECTIONS must be the last entry in this structure.

Members

Currently, only the following members are supported:

DW_SECTION_IS_DEBUG_DATA
Contains `.debug_*` section information.

DW_SECTION_IS_REL
Contains `.rel.debug_*` section information.

DW_SECTION_IS_RELA
Contains `.rela.debug_*` section information.

dwarf_debug_section operation

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

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 object.

type
Input. This accepts the debug-section type.

content
Input. This accepts the debug-section content.

ret_section
Output. This returns the `Dwarf_Section` object.

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

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-section traversal consumer operations

These operations are used for traversing DIEs within `DIE-format` sections.

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

The `dwarf_nthdie` operation locates the nth DIE of a given DIE-format section unit.

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. Please 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);
}
```

DIE-query consumer operations

These operations look 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.

dwarf_diecount operation

The `dwarf_diecount` operation counts the number of children DIEs in the debug-section unit for a given parent DIE.

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 returns the DIE count for a unit.

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

dwarf_dieindex operation

The `dwarf_dieindex` operation looks for the DIE index within a section unit.

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.

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.

DIE-search consumer operations

These operations search within the libdwarf object for one or more DIEs, when given one or more of its properties.

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_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);
```

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_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-attribute query consumer operation

This operation looks for specific information about a particular DIE attribute.

dwarf_attr_offset operation

The `dwarf_attr_offset` operation returns the section offset within that attribute's DIE.

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: If the `die` and the `attr` values are not related, the result is meaningless.

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. Counter-location operations use two data types: `Dwarf_PC_Locn` and `Dwarf_Subpgm_Locn`.

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."

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.

Consumer Flag operations

DWARF consumer flag operations test or set DWARF flag bits.

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.

Chapter 3. Program Prolog Area (PPA) extensions to DWARF consumer APIs

The Program Prolog Area (PPA) blocks are data areas in DWARF consumer APIs that conform to the Language Environment run-time conventions.

PPA blocks are generated by a language translator, which may be either of the following:
- 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 extension

This section discusses the PPA debug section, which is an IBM extension.

.debug_ppa

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 is an example of 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" on page 2.

PPA DIEs and attributes

The `.debug_ppa` section is an IBM extension. It provides the Debug Information Entries (DIEs) that describe the PPA blocks in each application executable module.

These DIEs describe:
- The hierarchy of PPA1 (subprogram) blocks for each PPA2 (CU) block
- The address of each PPA1 and PPA2 block within the application executable module
- The offset of the CU header within the `.debug_info` section which corresponds to the PPA2 block
- The relative offset (within that CU-level portion of the `.debug_info` section) for the subprogram symbol DIE which corresponds to each PPA1 block

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.

Each `.debug_ppa` section is organized as follows:
- Block header
- Section-specific DIEs
- Reference section

Each `.debug_ppa` section also has associated sections.

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. That header describes a block of information in the `.debug_info` section, as contributed by a single CU.

The PPA block header contains:
- A 4-byte or 12-byte unsigned integer representing the length of of the `.debug_ppa` block, not including the length of the field itself

  Note: 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.

- A 2-byte unsigned integer representing the version of the DWARF information for that block of `.debug_ppa` information
- 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

Note: For more information about the block header, refer to DWARF Debugging Information Format Standard, V3, Draft 7.

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:

- 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.

.rela.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` is an offset into the `.debug_ppa` section. It consists of a 4-byte or an 8-byte value, which is the offset from the beginning of the section to the first byte of the Debug Information Entry (DIE) being referenced.

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 4. Program source extensions to DWARF consumer APIs

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 block is generated for each CU:
- For each primary source file
- For each include file

Each .debug_srcfiles section block may share the associated .debug_abbr section block, but must have a separate .rela.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.

Opcodes that are related to IBM's source-file extensions include the following:

- DW_LNE_IBM_define_global_file opcode refers to the source-file entry defined in .debug_srcfile debug section
- DW_LNE_IBM_set_system_flag opcode sets the source-line attributes.
- DW_LNE_IBM_clear_system_flag opcode clears all the source-line attributes.

Block header

Each block of information in the .debug_srcfiles section will begin with a header, which consists of the following information:

Block length
: A 4-byte or 12-byte unsigned integer represents the length of of the .debug_srcfiles block. 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 contain the actual length represented as an unsigned 64-bit integer.

DWARF version
: A 2-byte unsigned integer represents the DWARF version of the .debug_srcfiles information for the block. For DWARF Version 3, the value for this field is 2.

.debug_abbrev offset
: A 4-byte or 8-byte unsigned offset into the .debug_abbrev section. This offset associates the source file information with a particular set of DIE abbreviations.
  - In the 32-bit DWARF format this is a 4-byte unsigned length.
  - In the 64-bit DWARF format, this is an 8-byte unsigned length.

The source-file information-block header is used to navigate the information blocks in the .debug_srcfiles section. It is similar in format and intent to the standard Compile-Unit Header; the CU header describes a block of information in the .debug_info section as contributed by a single CU. For more information, see section 7.5.1 in DWARF Debugging Information Format, V3, Draft 7.

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.

.rela.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_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.

Source-file entries

DW_TAG_src_location DIE tag

The DIE with the tag DW_TAG_IBM_src_location identifies the system and primary location of the source file. For captured source, this will be the location of the original source file at the time of program translation (preprocessing, compilation or assembly). 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

DW_TAG_src_file DIE tag

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_language

- 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_language
: This attribute value is a constant of form DW_FORM_data*, and it is a standard DWARF attribute. In this context, it defines the primary source language translator used to process the program source file.

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.

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 is an src file parameter of form DW_FORM_data4 or DW_FORM_data8. This is 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

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 desired 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 generate views of source files.

A developer can access the source files either directly or through a predefined view. Direct access is done through the DIEs in the .debug_srcfiles section. The basic-source view (DW_SRV_basic_source) shows the original program source, as provided by the user.

Developers can also create their own views by using DW_SRV_user_defined as the view name.

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 5. DWARF expression operations

The IBM extensions to DWARF expression operations 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 2.4 in *DWARF Debugging Information Format*, V3, Draft 7.

Defaults and general rules

The following are defaults and general rules 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.
- Decimal, complex and user types always require full information.
- If the previous expression in the expression evaluator emits a user type, then the next expression must take either a user type or a conversion routine that will convert the user type to the required type.

DW_OP_IBM_conv operation

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 second set of operands indicates the new type.
- Both types will be encoded using the minimum amount of information required to define the type.
- Each type description may take up to four operands to describe 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: Type conversion

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	Additional Parameters
DW_ATE_signed_char or DW_ATE_unsigned_char	No additional parameters.
DW_ATE_address, DW_ATE_boolean, DW_ATE_float, DW_ATE_IBM_float_hex, DW_ATE_signed or DW_ATE_unsigned	**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_complex_float or DW_ATE_IBM_complex_float_hex	**Container size** A 2-byte unsigned integer giving the complete size of the complex type in bytes. The complex number may not include padding. **element size** A 1-byte unsigned integer giving the size of each element in bytes.
DW_ATE_IBM_packed_decimal or DW_ATE_IBM_zoned_decimal	**Container size** A 2-byte unsigned integer, with the size of the value including any padding and sign bits. This value is in bytes. **Total number of digits** A 1-byte unsigned integer with the number of digits of the stored number. **Decimal Places** A 1-byte unsigned integer with the number of digits after the implied decimal point.

Table 5. DW_OP_IBM_conv parameters (continued)

Base Type	Additional Parameters
DW_ATE_IBM_user_type	This parameter must have three additional parameters which will be interpreted by the user-supplied override conversion routine. The format and order of the parameters is as follows: 1. A 2-byte unsigned integer 2. A 1-byte unsigned integer 3. A 1-byte unsigned integer
DW_ATE_IBM_register	This is not valid on a `DW_OP_IBM_conv` operation and will cause an error.

DW_OP_IBM_builtin operation

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_Op` 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.

Table 6. DW_OP_IBM_builtin functions (continued)

Sub Op	**Description**
DW_SubOP_builtin_substr (0x02)	This `Sub_Op` 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_Op` 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.
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`

Table 7. DW_OP_IBM_prefix additional parameters (continued)

Prefix	Description
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.

Chapter 6. Program line-number extensions to DWARF consumer APIs

DWARF 3 provides program-line and statement information. One of the characteristics of the DWARF3 format is the manner in which program-line breakpoints are processed. This format enables a DWARF-compliant debugging program to set line breakpoints by using special instruction sequences that have been generated in advance by the compiler.

Each of these instruction sequences includes:
- An execute (EX) statement
- An offset for each hook type

Some compilers generate special-instruction sequences prior to program execution to indicate breakpoint positions. For example, the IBM z/OS C/C++ compiler generates special-instruction sequences that include an EX (execute) opcode, as follows:
- The initial EX target instruction is an NOP, so the hook is always inactive when the program starts. A hook is activated when the target of the EX opcode is replaced with another suitable instruction.
- As program events occur, the debugger consults the line table in the `.debug_line` section to determine whether a particular hook event is processed or ignored. Execution of the EX opcode target at any hook site of this type:
  - Saves the current program state
  - Passes program control to the debugger.
- Normal execution resumes at the next application program instruction.

CDA debug hooks

IBM program line-number extensions provide system-specific line-number attributes that a debugger can use to identify debug hook types. These attributes are listed in the following table.

Table 8. Line-number attributes

Attribute	**Description**
DW_SAT_IBM_hook	An unknown hook.
DW_SAT_IBM_path_label	The path label.
DW_SAT_IBM_path_call_return	The Path: call after return from call.
DW_SAT_IBM_alloc	The storage allocation.
DW_SAT_IBM_autoinit	Automatic initialization.
DW_SAT_IBM_path_do_begin	The start of a do..while loop.
DW_SAT_IBM_path_true_if	An if statement evaluated to true.
DW_SAT_IBM_path_false_if	An if statement evaluated to false.
DW_SAT_IBM_path_when_begin	The start of a switch/case for a specific case.
DW_SAT_IBM_path_otherwise	The start of a switch/case for the default case.
DW_SAT_IBM_path_postcompound	The merge of multiple paths.

Table 8. Line-number attributes (continued)

Attribute	Description
DW_SAT_IBM_path_call_begin	A function call, after the parameter-list build, but before the actual call.
DW_SAT_IBM_goto	A goto statement.
DW_SAT_IBM_block_exit	The current instruction at the exit of a block.
DW_SAT_IBM_multiexit	The scope of a block of multiexits.
DW_SAT_IBM_prologue_begin	The beginning of a function prologue.

Line-number program

In the `.debug_line` section, the line data associated with each CU is stored as a matrix, encoded in a line-number program. When line data for a CU is required, the line-number program state machine regenerates it, using the original matrix as input.

The state machine has the following registers:

address
The PC value corresponding to a machine instruction generated by the compiler.

file
An unsigned integer indicating the identity of the source file corresponding to a machine instruction.

line
An unsigned integer indicating a source line number.

column
An unsigned integer indicating a column number within a source line.

is_stmt
A Boolean flag indicating that the current instruction is the beginning of a statement.

basic_block
A Boolean flag indicating that the current instruction is the beginning of a basic block.

end_sequence
A Boolean flag indicating that the current address is that of the first byte after the end of a sequence of target machine instructions.

prologue_end
A Boolean flag indicating that the current address is one where execution is suspended for an entry breakpoint of a function.

epilogue_begin
A Boolean flag indicating that the current address is one where execution is suspended for an exit breakpoint of a function.

A line-number program begins with a line-number program header, which has:

- The parameter line_base field, which specifies the minimum value that a special opcode can add to the line register. line_range field defines the range of values it can add to the line register.

- The parameter line_range field, which defines the range of values a special opcode can add to the line register.
- A list of include directory path names. These may be absolute paths, or relative to the CU current directory.
- An entry for each source file that contributed to the statement information. Each entry has:
  - The include directory index
  - The timestamp of the most recent file modification
  - The file length

Debug sections

For more information about .debug_line, please refer to section 6.2 in DWARF Debugging Information Format, V3, Draft 7.

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_srclines`
- `dwarf_access_lineinfo`

Type definition

```
typedef struct Dwarf_Line_s* Dwarf_Line;
```

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_linepgm_owner operation

The dwarf_linepgm_owner operation locates the owner of the line-number program.

dwarf_linepgm_owner looks for the DIE where the DW_AT_stmt_list attribute matches the linepgm_ofs variable. The DIE is a either a root or a subprogram DIE. If it is a subprogram DIE, then the operation sets ret_subpgm_die to the DIE, and sets ret_root_die to the root DIE. If the DIE a root DIE, this operation sets ret_root_die to the DIE, and ret_subpgm_die to NULL.

Prototype

```
int dwarf_linepgm_owner (
    Dwarf_Debug          dbg,
    Dwarf_Off            linepgm_ofs,
    Dwarf_Die*           ret_root_die,
    Dwarf_Die*           ret_subpgm_die,
    Dwarf_Error*         error);
```

Parameters

dbg
Input. This accepts a libdwarf consumer object.

linepgm_ofs
Input. This accepts the offset for the line-number program.

ret_root_die
Output. This returns the DIE that owns the line-number program.

ret_subpgm_die
Output. This returns the subprogram DIE that owns the line-number program, if it is a subprogram.

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

Return values

The dwarf_linepgm_owner operation returns the root DIE for either a CU or a partial unit.

If the given line-number program is a sublevel program, then this operation returns subprogram DIE. If linepgm_ofs is not a valid line program offset, then it returns DW_DLV_NO_ENTRY.

dwarf_subpgm_linepgm operation

The dwarf_subpgm_linepgm operation determines whether the line-number program is at a subprogram level.

Prototype

```
int dwarf_subpgm_linepgm(
    Dwarf_Debug              dbg,
    Dwarf_Off                linepgm_ofs,
    Dwarf_Bool*              returned_bool,
    Dwarf_Error*             error);
```

Parameters

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

linepgm_ofs
Input. This accepts the offset for the line-number program.

returned_bool
Output. This returns the test results.

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

Return values

The `dwarf_subpgm_linepgm` operation returns `DW_DLV_NO_ENTRY` if `linepgm_ofs` is not a valid line-program offset.

dwarf_access_lineinfo operation

The `dwarf_access_lineinfo` operation decodes a line-number program into the line-number information matrix, and returns a list of line-number matrix rows.

Prototype

```
int dwarf_access_lineinfo(
     Dwarf_Debug            dbg,
     Dwarf_Off              linepgm_ofs,
     Dwarf_Line**           linebuf,
     Dwarf_Signed*          linecount,
     Dwarf_Error*           error);
```

Parameters

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

linepgm_ofs
Input. This accepts the offset for the line-number program.

linebuf
Output. This returns a list of line numbers for the matrix rows.

linecount
Output. This returns a count of the `linebug` list.

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

Return values

The `dwarf_access_lineinfo` operation returns `DW_DLV_NO_ENTRY` if `linepgm_ofs` is not a valid line-program offset.

dwarf_line_subline operation

The `dwarf_line_subline` operation searches for the source subline number for a line-matrix row.

Prototype

```
int dwarf_line_subline(
    Dwarf_Line           line,
    Dwarf_Unsigned*      ret_subline,
    Dwarf_Error*         error);
```

Parameters

line
Input. This accepts a line number of a matrix row.

ret_subline
Output. This returns the sub-line number.

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

dwarf_line_viewidx operation

The `dwarf_line_viewidx` operation searches for the source-view index for a line-matrix row.

Prototype

```
int dwarf_line_viewidx(
    Dwarf_Line                   line,
    Dwarf_IBM_src_view*          ret_view_idx,
    Dwarf_Error*                 error);
```

Parameters

line
Input. This accepts a line number of a matrix row.

ret_view_idx
Output. This returns the view index.

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

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_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. Global lookup tables and the DWARF consumer operations that use them

The following topics provide information about operations that use global lookup tables to expedite access to debugging information. DWARF consumer operations access the Dwarf_section_type object, using the name lookup tables. For more information about lookup tables, refer to section 6.1 in *DWARF Debugging Information Format, V3, Draft 7*.

For a description of DWARF debugging sections, see "Dwarf_section_type object" on page 14.

Debug sections

Debug sections store the contents of global lookup tables.

debug_pubnames
Stores global objects and functions.

.debug_pubtypes
Stores file-level types.

debug_funcnames
Stores file-static functions. This is a MIPS extension, not an IBM extension.

debug_varnames
Stores file-static data symbols. This is a MIPS extension, not an IBM extension.

.debug_weaknames
Stores weak symbols. This is a MIPS extension, not an IBM extension.

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_AUTOMT      =  18,
  |                         DW_SECTION_NUM_SECTIONS
|                       } Dwarf_section_type;
```

Note: DW_SECTION_NUM_SECTIONS must be the last entry in this structure.

Members

The following members are supported for the global lookup table functions:

DW_SECTION_DEBUG_PUBNAMES
Contains .debug_pubnames information.

DW_SECTION_DEBUG_PUBTYPES
Contains .debug_pubtypes information.

DW_SECTION_DEBUG_FUNCNAMES
Contains .debug_funcnames information.

DW_SECTION_DEBUG_VARNAMES
Contains .debug_varnames information.

DW_SECTION_DEBUG_WEAKNAMES
Contains .debug_weaknames information.

DW_SECTION_NUM_SECTIONS
This contains the number of sections in the structure. It must always be the last entry in this structure.

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 desired 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 (...).

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 8. 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_fpr2         = 17,
   DW_FRAME_390_fpr4         = 18,
   DW_FRAME_390_fpr6         = 19,
   DW_FRAME_390_fpr1         = 20,
   DW_FRAME_390_fpr3         = 21,
   DW_FRAME_390_fpr5         = 22,
   DW_FRAME_390_fpr7         = 23,
   DW_FRAME_390_fpr8         = 24,
   DW_FRAME_390_fpr10        = 25,
   DW_FRAME_390_fpr12        = 26,
   DW_FRAME_390_fpr14        = 27,
   DW_FRAME_390_fpr9         = 28,
   DW_FRAME_390_fpr11        = 29,
   DW_FRAME_390_fpr13        = 30,
   DW_FRAME_390_fpr15        = 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,
   DW_FRAME_390_CEESTART   = 67,
   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_CEESTART
Load-module address.

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 87.

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 9. 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.

Chapter 10. 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 11. 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 12. 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 13. 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_reference_with_reloc operation

The `dwarf_add_AT_reference_with_reloc` operation adds the attributes of a reference class.

These attributes refer to other CU references. That is, relocation entries will be generated. 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_with_reloc (
    Dwarf_P_Debug       dbg,
    Dwarf_P_Die         ownerdie,
    Dwarf_Half          attr,
    Dwarf_P_Die         otherdie,
    Dwarf_Error*        error);
```

Parameters

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

ownerdie
Input. This accepts the owning DIE.

attr
Input. This accepts the DIE attribute.

otherdie
Input. This accepts the referenced DIE. It is referenced by `ownerdie`.

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

Return values

The `dwarf_add_AT_reference_with_reloc` operation returns the `Dwarf_P_Attribute` descriptor for `attr` on success, and `DW_DLV_BADADDR` if:

- `DW_DLV_OK` if successful
- `DW_DLV_ERROR` if:
  - `dbg` is NULL
  - `ownerdie` is NULL
  - `otherdie` is NULL
  - Attribute does not fall under the reference class

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.

Chapter 14. 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_add_global_view_decl operation

The `dwarf_add_global_view_decl` operation adds a global source-view declaration.

It results in a `DW_LNE_IBM_define_source_view` opcode in the body of the current line-number program.

Prototype

```
int dwarf_add_global_view_decl(
    Dwarf_P_Debug        dbg,
    Dwarf_P_Die          view_die,
    Dwarf_Unsigned *     ret_view_idx,
    Dwarf_Error*         error);
```

Parameters

dbg
: Input. This accepts a `libdwarf` producer object.

view_die
: Input. This accepts the source-view DIE object in the `.debug_srcviews` section.

ret_view_idx
: Output. This returns the source-view index.

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

Return values

The `dwarf_add_global_view_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_view_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 15. 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

Chapter 16. Fast-access (name) producer operation

The API in this section creates 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 the type of 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 `DW_DLV_ERROR` if:

- `dbg` is NULL
- `die` is NULL
- `pubname` is NULL

Chapter 17. 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 18. Range-list producer APIs

Range-list producer operations update the `.debug_ranges` section.

Debug sections

For information about range-list debug sections, see section 2.16.3 in the DWARF Debugging Information Format Standard, V3, Draft 7.

PPA DIEs and attributes

The `.debug_ppa` section is an IBM extension. It provides the Debug Information Entries (DIEs) that describe the PPA blocks in each application executable module.

These DIEs describe:

- The hierarchy of PPA1 (subprogram) blocks for each PPA2 (CU) block
- The address of each PPA1 and PPA2 block within the application executable module
- The offset of the CU header within the `.debug_info` section which corresponds to the PPA2 block
- The relative offset (within that CU-level portion of the `.debug_info` section) for the subprogram symbol DIE which corresponds to each PPA1 block

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.

Each `.debug_ppa` section is organized as follows:

- Block header
- Section-specific DIEs
- Reference section

Each `.debug_ppa` section also has associated sections.

Attribute form extension

The DW_TAG_ranges extension has an attribute value of class `rangelistptr`.

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 19. 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 20. 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 utilities" on page 137

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.

```
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 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

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size

Using assistive technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using such products to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to *z/OS TSO/E Primer*, *z/OS TSO/E User's Guide*, and *z/OS ISPF User's Guide Vol I* for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

z/OS information

z/OS information is accessible using screen readers with the BookServer/Library Server versions of z/OS books in the Internet library at:

`http://www.ibm.com/servers/eserver/zseries/zos/bkserv/`

Notices

This information was developed for products and services offered in the U.S.A.

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
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, 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.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites 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.

Programming interface information

This publication documents intended Programming Interfaces that allow the customer to write programs to obtain services of Common Debug Architecture.

Trademarks

Other company, product, and service names may be trademarks or service marks of others.

Standards

The `libddpi` library supports the DWARF Version 3 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 working draft 7 of the DWARF 3 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

Program Number: 5694-A01

Printed in USA

SC09-7655-03