

z/OS



Common Debug Architecture User's Guide

z/OS



Common Debug Architecture User's Guide

Note

Before using this information and the product it supports, read the information in “Notices” on page 61.

Third Edition (September 2008)

This edition applies to Run-Time Library Extensions in Version 1 Release 10 of z/OS (5694-A01) and to all subsequent releases until otherwise indicated in new editions. This edition replaces SC09-7653-01. Ensure that you use the correct edition for the level of the program listed above. Also, 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.

© Copyright International Business Machines Corporation 2004, 2008. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

	About this document	v
	Who should use this document	v
	A note about examples	vi
	CDA and related publications	vi
	Softcopy documents	viii
	Softcopy examples	viii
	Where to find more information	viii
	Run-Time Library Extensions on the World Wide Web	viii
	Information updates on the web	viii
	Technical support	viii
	How to send your comments	ix
	Chapter 1. About Common Debug Architecture	1
	CDA libraries and utilities	2
	libelf	2
	libdwarf	3
	libddpi	3
	isdcnv	6
	dwarfdump	6
	Changes for CDA in z/OS V1R10	6
	CDA requirements and recommendations	7
	CDA limitations	7
	Chapter 2. Overview of reading and writing CDA debugging information	9
	Creating an ELF descriptor	9
	Writing DWARF data to the ELF object file	12
	Reading from an ELF object file with libelf and libdwarf	14
	Reading from an ELF object file with libelf, libdwarf, and libddpi	15
	Accessing debugging information	18
	Accessing z/OS XL C/C++ debugging information	19
	Accessing and converting ISD information	19
	Accessing non-z/OS XL C/C++ debugging information	20
	Chapter 3. Using consumer functions	21
	Initializing libelf and libdwarf	21
	Steps to relocate addresses within an ELF file	22
	Example: Relocating addresses within an ELF file	23
	Consuming DWARF data	36
	Traversing the DIE hierarchy	37
	Accessing information in a DIE	37
	Terminating libelf and libdwarf	38
	Chapter 4. Using producer APIs	39
	Steps for creating a line-number table	39
	Steps for preparing the debug_ppa section	39
	Steps for converting application symbols	40
	Additional steps	40
	Location expressions	41
	Example: Converting symbol attributes into DIEs	41
	Chapter 5. Using consumer and producer functions	43
	Creating a consumer application with ISD conversion functionality	43
	Initializing the libddpi environment	44

Creating and using DWARF consumer objects	45
Terminating the DWARF and ELF objects	46

Chapter 6. In Storage Debug (ISD) Information Conversion Utility. 47

Chapter 7. Using the module map to improve performance	51
APIs that support use of the module map	51
Sample statements that illustrate use of a module map	53

Appendix A. Diagnosing problems	57
Using the diagnosis checklist.	57
Avoiding installation problems	58

Appendix B. Accessibility	59
Using assistive technologies	59
Keyboard navigation of the user interface	59
z/OS information	59

Notices	61
Programming interface information.	62
Trademarks	62
Standards.	63

Bibliography	65
z/OS Run-Time Library Extensions	65
z/OS.	65
z/OS XL C/C++.	65
z/OS Language Environment.	65
z/Architecture	66

Index	67
------------------------	-----------

About this document

This information introduces the user to Common Debug Architecture (CDA), which is part of the IBM® z/OS® Run-Time Library Extensions element. This document first provides a high-level overview of CDA. The document then illustrates how to use the CDA libraries and utilities, through explanations and examples that build on each other. Finally, it shows an example implementation, using the utilities that are shipped with CDA.

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.

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 z/OS operating system. The libraries provided by CDA allow applications to create or query DWARF debugging information from ELF object files on the z/OS operating system.

Purpose

This document is provided as 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 C++ on z/OS
- POSIX on z/OS
- The IBM z/OS Language Environment® (LE)
- z/OS UNIX® System Services (USS) shell

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
<i>z/OS Common Debug Architecture Library Reference, SC09-7654</i>	The reference for IBM's <code>libddpi</code> library. It includes: <ul style="list-style-type: none">• 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/ .
<i>z/OS DWARF/ELF Extensions Library Reference, SC09-7655</i>	The reference for IBM extensions to the <code>libdwarf</code> and <code>libelf</code> libraries. It includes: <ul style="list-style-type: none">• Extensions to <code>libdwarf</code> consumer APIs (Chapters 2 through 8)• Extensions to <code>libdwarf</code> producer APIs (Chapters 9 through 19)• Extensions to <code>libelf</code> APIs and utilities (Chapter 20) This document discusses only these extensions, and does not provide a detailed explanation of DWARF and ELF. See http://www.ibm.com/software/awdtools/libraryext/library/ .
<i>System V Application Binary Interface Standard</i>	The Draft April 24, 2001 version of the ELF standard. For more information, go to: http://www.ibm.com/software/awdtools/libraryext/library/ .
<i>ELF Application Binary Interface Supplement</i>	The Draft April 24, 2001 version of the ELF standard supplement. For more information, go to: http://www.ibm.com/software/awdtools/libraryext/library/ .
<i>DWARF Debugging Information Format, Version 3</i>	The Draft 8 (November 19, 2001) version of the DWARF standard. This document is available on the web.
<i>Consumer Library Interface to DWARF</i>	The revision 1.48, March 31, 2002, version of the <code>libdwarf</code> consumer library. See http://www.ibm.com/software/awdtools/libraryext/library/ .
<i>Producer Library Interface to DWARF</i>	The revision 1.18, January 10, 2002, version of the <code>libdwarf</code> producer library. See http://www.ibm.com/software/awdtools/libraryext/library/ .
<i>MIPS Extensions to DWARF Version 2.0</i>	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	<p>Guidance information for:</p> <ul style="list-style-type: none"> • 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 <p>See http://www.ibm.com/software/awdtools/czos/library.</p>
z/OS XL C/C++ Programming Guide, SC09-4767	<p>Guidance information for:</p> <ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • z/OS XL C/C++ User's Guide, SC09-4767 • z/OS XL C/C++ Programming Guide, SC09-4765
General discussion of CDA	<ul style="list-style-type: none"> • 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)	<ul style="list-style-type: none"> • z/OS Common Debug Architecture Library Reference, SC09-7654
A guide to using the libraries	<ul style="list-style-type: none"> • z/OS Common Debug Architecture Library Reference, SC09-7654
Examples of producer and consumer programs	<ul style="list-style-type: none"> • 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: <http://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 <http://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*.

Softcopy examples

The example shown in “Initializing the libddpi environment” on page 44, described in Chapter 5, “Using consumer and producer functions,” on page 43, is available in the directory `/usr/lpp/cbc1ib/source`.

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 IBM 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:
<http://www.ibm.com/software/awdtools/czos/support>.

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:
<http://www.ibm.com/software/awdtools/czos/>

| **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.

CDA libraries and utilities

CDA comprises three libraries and two utilities.

The libraries are:

- `libelf`
- `libdwarf`
- `libddpi`

The utilities are:

- `isdcnvt`
- `dwarfdump`

To ensure compatibility, the `libdwarf` and `libelf` libraries are packaged together in a single DLL.

The `libddpi` library is available as a dynamic linking library, as well as an static library in XPLINK object form. It is highly recommended that you use the DLL version of `libddpi` when creating your application.

Regardless of whether a 64-bit or 31-bit version of a library is used, the created information is binary-equivalent. For example, the z/OS XL C/C++ compilers could use a 31-bit version of `libdwarf` and `libelf` to create the debug information, although the z/OS dbx debugger always uses a 64-bit version of `libdwarf`, `libelf` and `libddpi` when reading the debug information.

The `libelf` and `libdwarf` libraries each consist of two sets of APIs:

- A set of producer APIs that help create ELF/DWARF
- A set of consumer APIs that help users of ELF/DWARF

The `libdwarf/libelf` DLL contains all producer and consumer APIs.

libelf

The `libelf` APIs are used to create the ELF descriptor. The descriptor is then used by other APIs to read from, and write to, the ELF object file.

`libelf` is packaged as part of a dynamic link library (DLL). Both the 31-bit version and the 64-bit version are packaged as part of `CEE.SCEERUN2`.

- For 64-bit applications, `libelf` is shipped in the `CDAEQED` DLL as part of `CEE.SCEERUN2`.
- For 31-bit applications, `libelf` is shipped in the `CDAEED` DLL as part of `CEE.SCEERUN2`.

When compiling an application that uses the `libelf` library, you must:

- Include `libelf.h`
- Bind the module with an appropriate side deck:
 - For 64-bit applications:
 - Bind with `CEE.SCEELIB(CDAEQED)` if you are using an IBM MVS™ file system
 - Bind with `/usr/lpp/cbclib/lib/libelfdwarf64.x` if you are using a hierarchical file system
 - For 31-bit applications, use either of the following:

- Bind with CEE.SCEELIB(CDAEED) if you are using an MVS file system
- Bind with /usr/lpp/cbc1ib/lib/libelfdwarf32.x if you are using a hierarchical file system

Note: IBM has extended the libelf library to support C/C++ on the z/OS operating system. These extensions enable the libelf library to be used in various environments without additional extensions. The generic interfaces provided by libelf are defined as part of the UNIX System V Release 4 ABI. For descriptions of the interfaces supported by libelf, refer to the following documents:

- *System V Application Binary Interface Standard*
- *DWARF/ELF Extensions Library Reference*

libdwarf

The libdwarf APIs:

- Create or read DWARF objects that include DWARF debugging information
- Use ELF descriptors to read from, and write to, the ELF object file

libdwarf is packaged as a dynamic link library (DLL). Both the 31-bit version and the 64-bit version are packaged as part of CEE.SCEERUN2:

- For 64-bit applications, libdwarf is shipped in the CDAEQED DLL
- For 31-bit applications, libdwarf is shipped in the CDAEED DLL

When compiling an application that uses the libdwarf library, you must:

- Include both libdwarf.h and dwarf.h (which are located in the /usr/lpp/cbc1ib/include/libdwarf directory)
- Bind the module with an appropriate side deck:
 - For 64-bit applications:
 - Bind with CEE.SCEELIB(CDAEQED) if you are using an MVS file system
 - Bind with /usr/lpp/cbc1ib/lib/libelfdwarf64.x if you are using a hierarchical file system
 - For 31-bit applications:
 - Bind with CEE.SCEELIB(CDAEED) if you are using an MVS file system
 - Bind with /usr/lpp/cbc1ib/lib/libelfdwarf32.x if you are using a hierarchical file system

Note: IBM has extended the libdwarf library to support C/C++ on the z/OS operating system. The IBM extensions to libdwarf provide:

- Improved speed and memory utilization
- Support for the IBM zSeries eServer C/C++ languages

For information that is specific to these extensions, see *DWARF/ELF Extensions Library Reference*.

libddpi

The Debug Data Program Information library (libddpi) provides a repository for gathering information about a program module. A debugger or other program analysis application can use the repository to collect and query information from the program module.

libddpi:

- Supports conversion of non-DWARF C/C++ debugging information to the DWARF format. For example, the `libddpi` library is used to convert In Store Debug (ISD) information.
- Puts an environmental context around the DWARF information for both the producer APIs and the consumer APIs. For library reference information on `libddpi`, see *Common Debug Architecture Library Reference*.

The `libddpi` library is packaged as the static library `libddpi.a` in the `/usr/lpp/cbc/lib/lib` directory. This directory contains both the 31-bit and 64-bit versions of the library.

The `libddpi` library is also packaged as a dynamic link library (DLL). Both the 31-bit version and the 64-bit version are packaged as part of CEE.SCEERUN2:

- For 64-bit applications, `libddpi` is shipped in the CDAEQDPI DLL.
- For 31-bit applications, `libddpi` is shipped in the CDAEDPI DLL.

When creating or compiling an application that uses `libddpi`, you must:

- Include `libddpi.h` in your source code
The `libddpi.h` file is located in the `/usr/lpp/cbc/lib/include/libddpi/` directory.
- Bind the module with an appropriate side deck:
 - For 64-bit applications:
 - Bind with CEE.SCEELIB(CDAEQDPI) if you are using an MVS file system
 - Bind with `/usr/lpp/cbc/lib/lib/libddpi64.x` if you are using a hierarchical file system.
 - For 31-bit applications:
 - Bind with CEE.SCEELIB(CDAEDPI) if you are using an MVS file system.
 - Bind with `/usr/lpp/cbc/lib/lib/libddpi32.x` if you are using a hierarchical file system.

The main groups of APIs in `libddpi` are described in the following table:

API groups	Description
CDA application model APIs: <ul style="list-style-type: none"> • <code>Ddpi_Init</code> and <code>Ddpi_Finish</code> APIs • <code>Ddpi_Error</code> APIs • Processing storage deallocation APIs • <code>Ddpi_Addr</code> APIs • <code>Ddpi_Elf</code> loading API • <code>Ddpi_Info</code> APIs • <code>Ddpi_Space</code> APIs • <code>Ddpi_Process</code> APIs • <code>Ddpi_Thread</code> APIs • <code>Ddpi_Lock</code> APIs • <code>Ddpi_Mutex</code> APIs • <code>Ddpi_Cond</code> APIs • <code>Ddpi_Module</code> APIs • <code>Ddpi_Access</code> APIs • <code>Ddpi_Elf</code> APIs • <code>Ddpi_Class</code> APIs • <code>Ddpi_Section</code> APIs • <code>Ddpi_EntryPt</code> APIs 	This group of consumer and producer APIs allows developers to model applications they are analyzing and to use those models to keep track of debugging information.
CDA APIs that support use of the module map: <ul style="list-style-type: none"> • <code>Ddpi_Function</code> APIs • <code>Ddpi_Variable</code> APIs • <code>Ddpi_Type</code> APIs • <code>Ddpi_Source</code> APIs 	The operations in this group: <ul style="list-style-type: none"> • Find and extract information about a specific function, including static functions. Each <code>Ddpi_Function</code> object is owned by a <code>Ddpi_Elf</code> object. A <code>ddpi_function</code> operation queries one or more <code>Ddpi_Function</code> objects and extracts information about the specific function. • Provide information about global variables. Each <code>Ddpi_Variable</code> object is owned by a <code>Ddpi_Elf</code> object. • Provide information about external types. Each <code>Ddpi_Type</code> object is owned by a <code>Ddpi_Elf</code> object. • Provide information about source files. Each <code>Ddpi_Source</code> object is owned by a <code>Ddpi_Elf</code> object.
System-dependent APIs	This group provides system-specific helper APIs.
System-independent APIs	This group provides generic common helper APIs.
DWARF-expression APIs	This group provides a DWARF expression evaluator which assists with the evaluation of some of the DWARF opcodes.
Utilities	This group: <ul style="list-style-type: none"> • Helps convert ISD debugging information into DWARF debugging information. • supports the integrity of the program analysis application build.

isdcnvt

Note: isdcnvt cannot be used to convert 64-bit objects. Debug information for 64-bit XL C/C++ applications is available only in DWARF format.

isdcnvt is a stand-alone utility that converts objects with In Store Debug (ISD) information into an ELF object file. In other words, isdcnvt accepts objects with ISD C/C++ debugging information and generates an ELF object file containing debugging information in the DWARF format. It is shipped in the /usr/lpp/cbclib/bin/isdcnvt directory.

This converter supports debugging information generated by the TEST option for XL C/C++ compilers. For more information, see “CDA limitations” on page 7.

The following restrictions apply to the isdcnvt utility:

- Debugging information cannot be converted if the compilation unit (CU) has only line number information. This occurs if the GONUMBER and NOTEST compiler options are used.
- CUs cannot be converted if they have data only and do not contain any functions.

The required ISD information is generated by the IBM XL C/C++ compiler TEST option.

For more information on isdcnvt, see “Conversion APIs” in *Common Debug Architecture Library Reference*.

dwarfdump

The dwarfdump utility displays the debugging information of an ELF object file in user-readable form. It is shipped in the /usr/lpp/cbclib/bin directory.

dwarfdump works on DWARF objects nested within an ELF container. It can be used to validate the work of a developer who is accessing and manipulating DWARF debugging information.

Changes for CDA in z/OS V1R10

CDA libraries shipped with IBM z/OS V1R10 include additional APIs that support creation and use of a module map.

For detailed information about these APIs, see *Common Debug Architecture Library Reference*. For information about using these APIs, see Chapter 7, “Using the module map to improve performance,” on page 51.

The LIBELF_DLL_VERSION defined in libelf.h has been updated to 05002002 for the version of the DLL that contains these changes. To verify that the DLL version you are using is compatible with your code, always call the elf_dll_version function with the LIBELF_DLL_VERSION that is expected by the compiled code, as follows:

```
if (elf_dll_version(LIBELF_DLL_VERSION)) {  
    /* issue message to indicate incompatible dll's */  
}
```

If `elf_dll_version` returns a value other than zero, it indicates an attempt to use the code with an earlier version of ELF/DWARF support. Applications built with the current version of the ELF/DWARF DLL must use the current version of the ELF/DWARF runtime library.

In “Creating an ELF descriptor” on page 9, see the updated code for the following stages:

- Version Check
- Initialize Elf Descriptor

The `LIBDDPI_DLL_VERSION` defined in `libddpi.h` has been updated to 05002001. To verify that the DLL version you are using is compatible with your code, always call the `ddpi_dll_version` function with the `LIBDDPI_DLL_VERSION` that is expected by the compiled code, as follows:

```
if (ddpi_dll_version(LIBDDPI_DLL_VERSION)) {  
    /* issue message to indicate incompatible dll's */  
}
```

If `ddpi_dll_version` returns a value other than zero, it indicates an attempt to use the code with an earlier version of `libddpi`. Applications built with the current version of DDPI DLL must use the current version of `libddpi`.

In “Reading from an ELF object file with `libelf`, `libdwarf`, and `libddpi`” on page 15, see the updated code for the Initialize `libddpi` stage.

CDA requirements and recommendations

The CDA libraries are compiled with the z/OS XL C/C++ compiler.

To provide flexibility for developers who want to use the CDA application model, many `libddpi` objects have a variable-length user area. This allows the developers to store their own extra information in the `libddpi` model.

When you use CDA libraries, be aware of the following requirements and recommendations:

- If your application uses CDA libraries or utilities, you must compile it with the XPLINK compiler option.
- To ensure the best possible application performance, run applications with the `HEAPPOOLS(on)` run-time option.
 - For 31-bit applications, you must specify the `HEAPPOOLS(on)` option in a pragma or `CEEUOPT`.
 - For 64-bit applications, the `HEAPPOOLS(on)` option is the default.
- Notice the code set in which strings are accepted and returned. By default, most character strings accepted and returned by the CDA libraries are encoded in the ISO8859-1 code set. You can use code set conversion operations to change the code set. For more information about these operations, see *Common Debug Architecture Library Reference, SC09-7654*. For more information about the z/OS XL C/C++ compiler options, see *z/OS XL C/C++ User's Guide, SC09-4767*.

CDA limitations

When you use CDA libraries, be aware of the following limitations:

- Conversion support for ISD debugging information is available only for 31-bit object files, modules or program objects built with:

- IBM C/C++ for MVS/ESA™ V3R2
- Any release of z/OS XL C/C++

This support is not intended to work with debugging information generated by the IBM C/370™ or IBM AD/Cycle® C/370 compilers.

The CDA converter will be updated to match the TEST option support for the version of z/OS with which it is shipping. However, a lower-level CDA converter might not be able to properly convert the debugging data generated by the TEST option on a newer level of the z/OS C/C++ compiler.

- You must gather information and call the appropriate `libddpi` interface to generate objects (such as `Ddpi_Space` and `Ddpi_Process`) that can be used to model the behavior of an application under analysis. Although the `libddpi` library contains these objects, they are not created automatically when the application triggers an event.

Note: These `libddpi` objects were created to:

- Provide a structured information repository in a common format
- Allow CDA to use expanded queries across a whole application, whether or not the application information is in an ELF object file, or has been modelled using `libddpi` elements such as `Ddpi_Section`

Chapter 2. Overview of reading and writing CDA debugging information

This information discusses how the `libelf`, `libdwarf`, and `libddpi` libraries work together to access and use debugging information.

Reading and writing CDA debugging information

Note: This information requires that you are familiar with the concepts in Chapter 1, “About Common Debug Architecture,” on page 1 and the DWARF format. For more information about Debug Information Entries (DIEs) and their structure, see *DWARF Debugging Information Format*.

The information is divided up into the following sections:

Section	Description
“Creating an ELF descriptor”	This section explains how <code>libelf</code> uses a file handle and creates an ELF descriptor.
“Writing DWARF data to the ELF object file” on page 12	This section explains how <code>libelf</code> and <code>libdwarf</code> add debugging information from the ELF descriptor to the ELF object file.
“Reading from an ELF object file with <code>libelf</code> and <code>libdwarf</code> ” on page 14	This section explains how <code>libelf</code> and <code>libdwarf</code> use the debugging information from the ELF object file.
“Reading from an ELF object file with <code>libelf</code> , <code>libdwarf</code> , and <code>libddpi</code> ” on page 15	This section explains how <code>libelf</code> , <code>libdwarf</code> , and <code>libddpi</code> use the debugging information from the ELF object file.
“Accessing debugging information” on page 18	This section explains how to interrupt the consuming process in order to convert non-DWARF debugging information.

Creating an ELF descriptor

Producer and consumer operations use ELF descriptors to access ELF object files. The following diagram shows how an application uses the `libelf` library to create an ELF descriptor:

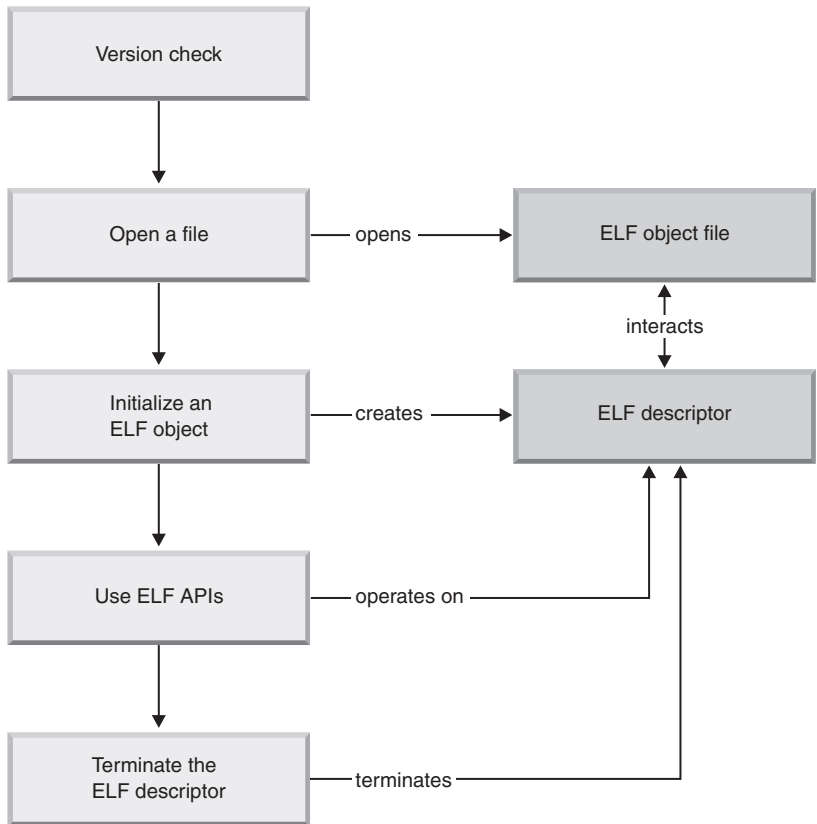


Figure 1. Creation of an ELF descriptor

Table 3 on page 11 describes how producer or consumer operations create an ELF descriptor with calls to `libelf` operations.

Table 3. Stages to create an ELF descriptor with calls to libelf operations

Stage	Description
Version check	<p>Since libelf is packaged as a DLL, this step will check the version. It is good practice to validate that the correct version of the DLL exists. For example:</p> <pre> #define _UNIX03_SOURCE #include <dldfcn.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 </pre> <p>It is mandatory to perform a verification of the ELF version before using the other functions offered by libelf. For example:</p> <pre> #include <dll.h> { /* Verify existence of libelf DLL */ dllhandle* dll_handle = dllload ("CDAEED"); if (dll_handle == NULL) { /* DLL not found, verify CEE.SCEERUN2 is in your STEPLIB */ } /* Verify that the current version of the ELF DLL meets or exceeds the minimum required version */ if (elf_dll_version (LIBELF_DLL_VERSION) != 0) { /* DLL version mismatch. - verify that "libelf.h" comes from: "/usr/lpp/cbclib/include/libelf" - verify CEE.SCEERUN2 is the first dataset on your STEPLIB - verify you have the latest service level of CDA libraries */ } } </pre>

Table 3. Stages to create an ELF descriptor with calls to libelf operations (continued)

Stage	Description
Version check (continued)	<p>It is mandatory to perform a verification of the ELF version before using the other operations offered by libelf. For example:</p> <pre> /* Verify that the current version of the ELF DLL meets or exceeds the minimum required version */ elf_version (EV_NONE); if (elf_version(EV_CURRENT) == EV_NONE) { /* libelf is out of date */ } </pre>
Open a file	<p>The producer or consumer operations create a file handle for the ELF object file. This file handle is used to create an ELF descriptor. Consult <i>z/OS XL C/C++ Run-Time Library Reference</i> for more information on opening files and creating file handles.</p>
Initialize ELF descriptor	<p>An ELF descriptor is required before you can call any other libelf operations. The file handle is used to initialize libelf and create an ELF descriptor for the ELF object file. The libelf operation that will create the ELF descriptor is determined by the operation that creates the file handle. For example, if the fopen operation creates the file handle, the elf_begin_b operation is used. The following code demonstrates how to use the file pointer obtained from fopen to create the ELF descriptor:</p> <pre> Elf* elf; /* ELF descriptor */ FILE* fp; /* File pointer */ /* Open test.dbg for reading */ fp = fopen ("test.dbg", "rb"); /* Create ELF descriptor for reading */ elf = elf_begin_b (fp, ELF_C_READ, NULL); </pre>
Operate on the descriptor	<p>After the ELF descriptor is initialized, you can call any libelf operations. For example, elf_getscn returns an ELF section, and elf_kind describes that section.</p>
Terminate ELF descriptor	<p>When the debugging information is no longer needed, the descriptor is terminated by the elf_end operation.</p> <p>Note: If you are using the libdwarf library, you must terminate its objects before you terminate the ELF descriptor. Close the file handle after the ELF descriptor is terminated.</p>

Writing DWARF data to the ELF object file

Once an ELF descriptor has been created, a producer application can use it to write DWARF debugging information to the ELF object file. This section discusses how a producer application writes to an ELF object file using the libelf and libdwarf libraries.

The following diagram shows an overview of the process.

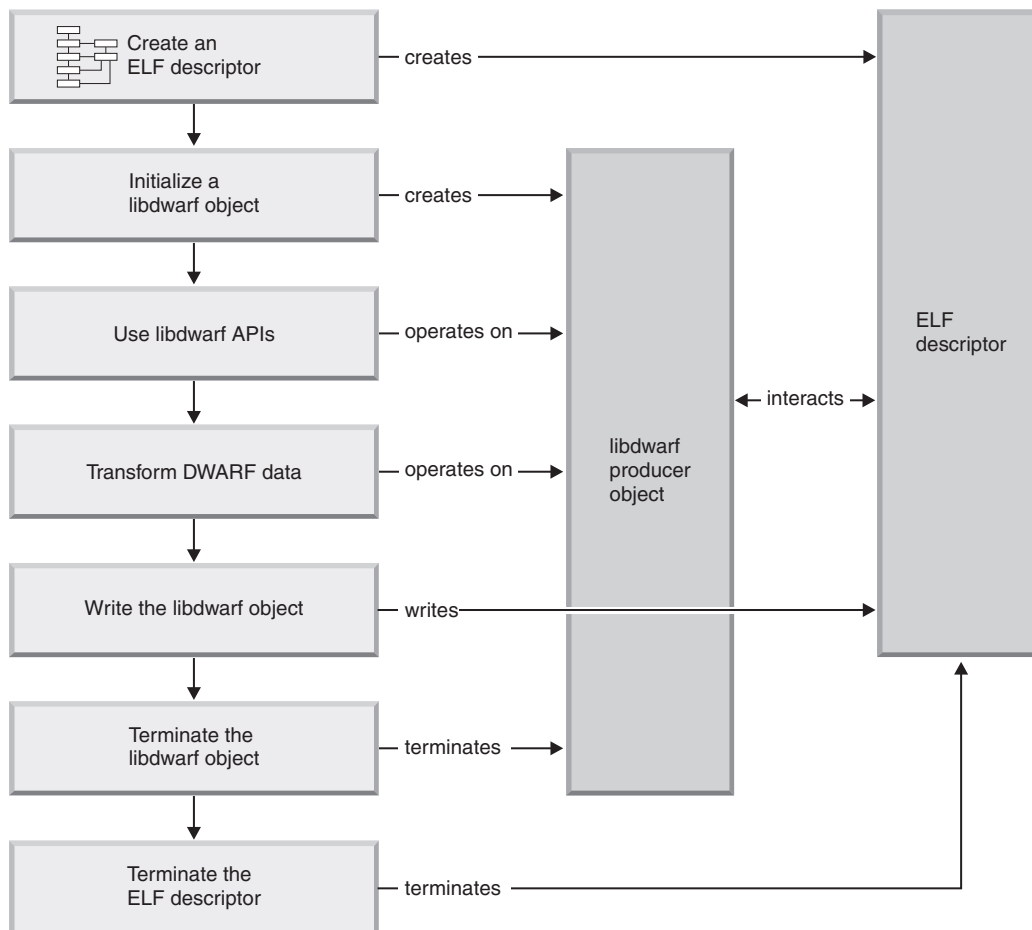


Figure 2. Write to an ELF object file

The following stages show how a producer application writes to an ELF object file with calls to `libelf` and `libdwarf` operations.

Stage	Description
Create an ELF descriptor	Create an ELF descriptor for writing. This descriptor will be used to write DWARF debugging information into the ELF object file. For more information, see “Creating an ELF descriptor” on page 9.

Stage	Description
Initialize a libdwarf object	<p>Initialize the Dwarf_P_Debug producer object. The object is initialized using the ELF descriptor. An ELF header (ehdr) is then created and used to complete the initialization.</p> <p>The following code demonstrates how to initialize the DWARF producer object:</p> <pre> Dwarf_P_Debug dbg; /* Producer DWARF object */ /* Initialize libdwarf producer instance */ flag = DW_DLC_WRITE DW_DLC_SIZE_32 DW_DLC_ISA_ELF_HDR DW_DLC_STREAM_RELOCATIONS; dbg = dwarf_producer_init_b(flag, /* callback function for creating ELF section*/ section_creation_func, /* error handling callback function*/ error_handling_func, /* arguments to be passed into error_handling_func*/ "error arguments", &dwarf_error); </pre> <p>Note: The ehdr is extracted from the descriptor. An update to the header will update the descriptor.</p> <pre> /* Create the ELF header */ ehdr = elf32_newehdr(elf); /* Initialize the ELF header */ ehdr->e_type = ET_REL; ehdr->e_machine = EM_S390; ehdr->e_version = EV_CURRENT; dwarf_producer_target(dbg, elf, &dwarf_error); </pre>
Use libdwarf APIs	libdwarf producer operations are called to add DWARF debugging information to the ELF object file. For example, dwarf_add_line_entry will add one line-number statement to the line number program matrix. dwarf_new_die will create a new DIE with a given DIE tag.
Transform DWARF data	dwarf_transform_to_disk_form must be called to format the DWARF debugging information before it can be written to the file. That is, the debugging information in the Dwarf_P_Debug object must conform to the actual binary representation of the ELF object file.
Write the libdwarf object	The data is written to the ELF object file by calling dwarf_producer_write_elf. libdwarf interacts with libelf to write all the gathered debug sections to the ELF object file that is managed by the ELF descriptor.
Terminate the libdwarf object	dwarf_producer_finish is called to terminate the Dwarf_P_Debug object.
Terminate the ELF descriptor	The ELF descriptor is terminated with elf_end.

Reading from an ELF object file with libelf and libdwarf

Once a descriptor has been created, consumer operations can use it to read the DWARF debugging information from the ELF object file. This section discusses how consumer operations read from an ELF object file using the libelf and libdwarf libraries.

The following diagram shows an overview of the process.

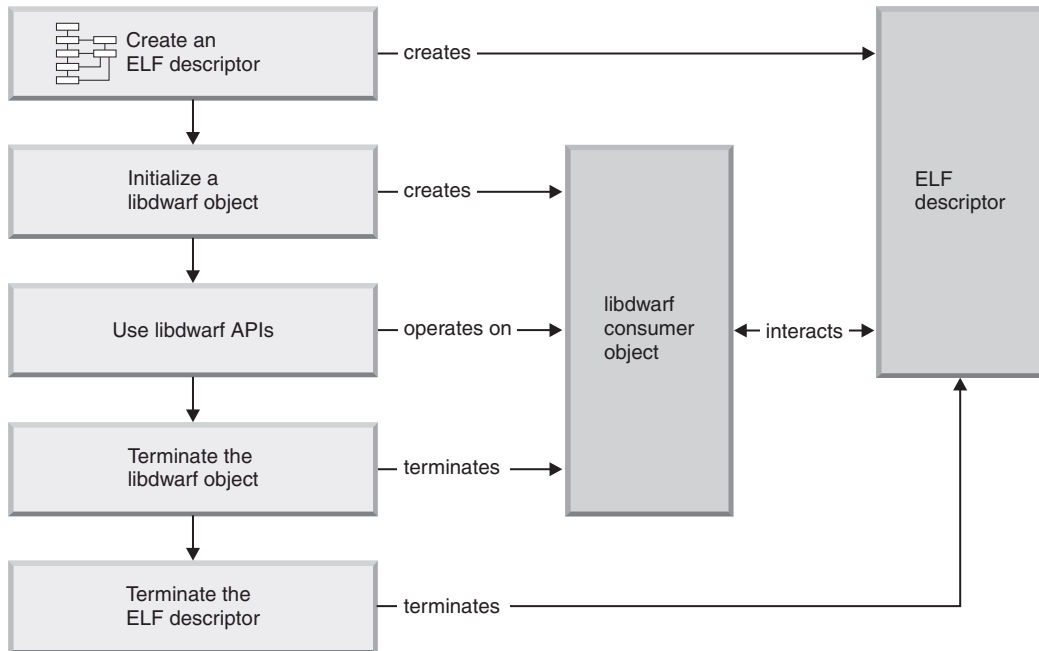


Figure 3. Read from an ELF object file with libelf and libdwarf

The following table shows the stages of reading from an ELF descriptor with calls to libelf and libdwarf operations.

Stage	Description
Create an ELF descriptor	Create an ELF descriptor for reading. This descriptor will be used to access the DWARF debugging information in the ELF object file. For more information, see “Creating an ELF descriptor” on page 9.
Initialize a libdwarf object	Initialize the Dwarf_Debug consumer object by calling dwarf_elf_init, using the ELF descriptor. libdwarf sets up the consumer libdwarf object to be able to load debugging information from the ELF descriptor.
Use libdwarf APIs	libdwarf operations are called to retrieve the DWARF data. For example, dwarf_get_globals will retrieve the list of global symbol entries, and dwarf_get_dies_given_name will return a list of DIEs in a section that match the given name.
Terminate the libdwarf object	dwarf_finish is called to terminate the Dwarf_Debug object.
Terminate the ELF descriptor	The ELF descriptor is terminated with elf_end.

Reading from an ELF object file with libelf, libdwarf, and libddpi

Once a descriptor has been created, consumer operations can use it to read the DWARF debugging information from the ELF object file. This section discusses how consumer operations reads from an ELF object file using the libelf, libdwarf, and libddpi libraries.

Note: The concepts in this section are based on “Reading from an ELF object file with libelf and libdwarf” on page 14.

The following diagram shows an overview of the process.

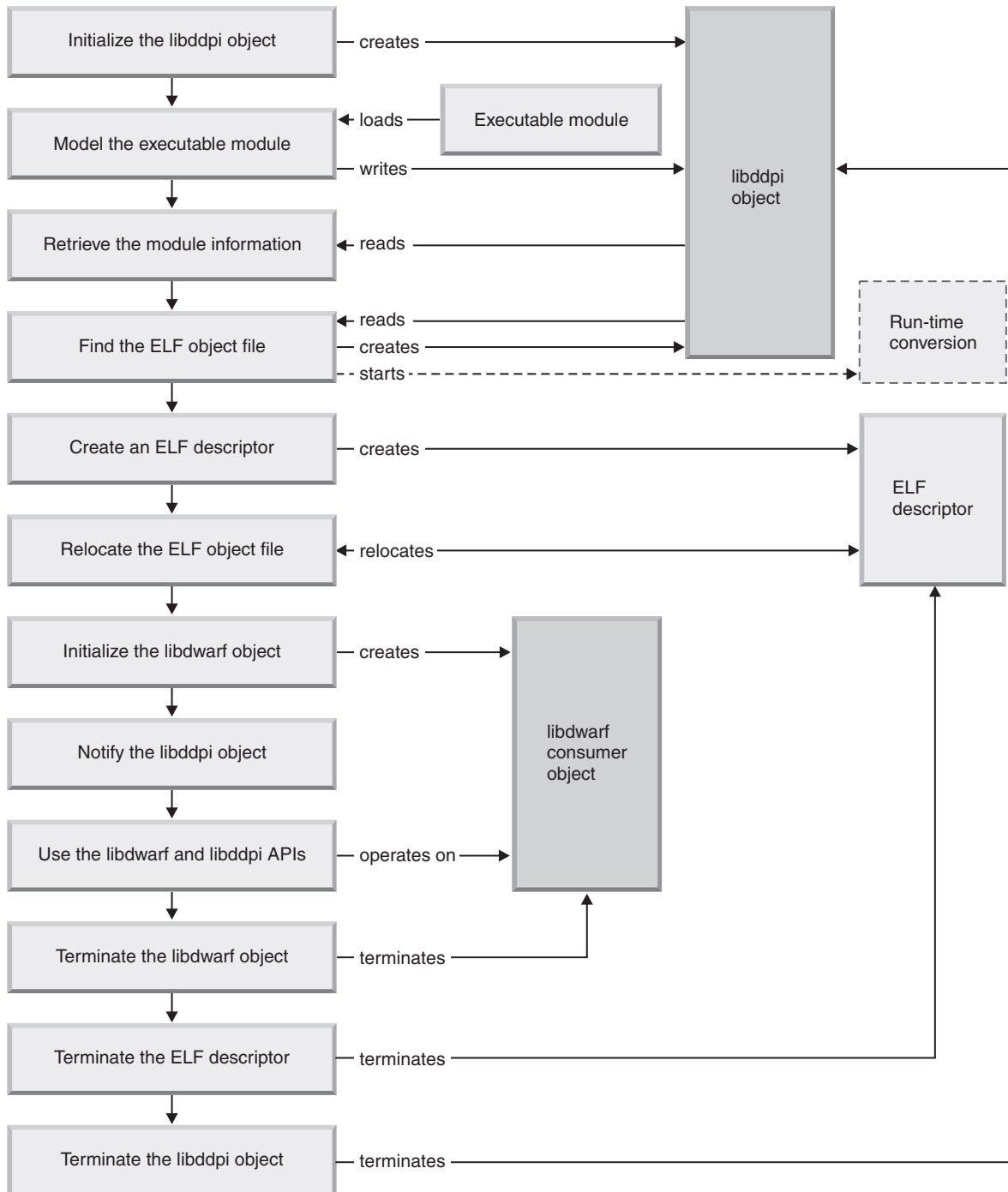


Figure 4. Read from an ELF object file with libelf, libdwarf, and libddpi

The following stages show how consumer operations read from an ELF object file using the libelf, libdwarf, and libddpi libraries.

Stage	Description
Initialize libddpi object.	<p>To validate the version of libddpi, use the following code:</p> <pre data-bbox="833 260 1453 1157"> #define _UNIX03_SOURCE #include <dldfcn.h> /* dlopen,dlsym */ #include "libddpi.h" void *cdadll; unsigned int (*version_chk)(unsigned int); unsigned int dll_version; #ifdef _LP64 #define __CDA_DDPI "CDAEQDPI" #else #define __CDA_DDPI "CDAEDPI" #endif #ifdef LIBDDPI_IS_DLL cdadll = dlopen(__CDA_DDPI, RTLD_LOCAL RTLD_LAZY); if (cdadll == NULL) { /* libddpi DLL not found */ } </dldfcn.h> version_chk = (unsigned int (*)(unsigned int)) dlsym(cdadll, "ddpi_dll_version"); if (version_chk == NULL) { /* Version API not found, should NEVER happen */ } dll_version = version_chk (LIBDDPI_DLL_VERSION); if (dll_version != 0) { /* Incompatible DLL version */ } dlclose(cdadll); #endif </pre> <p>Call <code>ddpi_init</code> to create a <code>Ddpi_Info</code> object. <code>Ddpi_Info</code> is a starting point that tracks:</p> <ul data-bbox="833 1255 1453 1360" style="list-style-type: none"> • The objects that model the application environment • The ELF object(s) • The DWARF object(s)
Model the executable module.	<p>Use libddpi operations to retrieve information from the executable module. For example:</p> <ul data-bbox="833 1444 1453 1696" style="list-style-type: none"> • <code>ddpi_space_create</code> represents the address space in which the executable module resides • <code>ddpi_storagelocn_create</code> provides access to user storage • <code>ddpi_module_create</code> represents the actual executable module • <code>ddpi_entrypt_create</code> represents the entry point of the executable module
Retrieve module information.	<p>Call <code>ddpi_module_extract_C_CPP_information</code>. This operation identifies all the compilation units (CU) in the executable module, then creates a <code>Ddpi_Elf</code> object to represent each CU. Each object holds the necessary information to load the DWARF debugging information that is in the CU.</p>

Stage	Description
Find ELF object file.	Call <code>ddpi_elf_get_elf_file_name</code> to search for the name of an ELF object file. If the file name can not be found it returns <code>DW_DLV_NO_ENTRY</code> , which indicates that this CU is not compiled with the <code>DEBUG(FORMAT(DWARF))</code> option. The debugging information may need to be converted to DWARF before calling any other CDA-compliant APIs. For more information, see “Accessing debugging information.”
Create ELF descriptor.	Open the ELF object file for reading and create an ELF descriptor. This descriptor will be used to access the DWARF debugging information in the ELF object file. For more information, see “Creating an ELF descriptor” on page 9
Relocate ELF object file.	Call <code>ddpi_elf_load_cu</code> to relocate the ELF object file. This ensures that the addresses within the file are the same as the addresses within the executable module. For more information, see “Steps for preparing the <code>debug_ppa</code> section” on page 39.
Initialize the <code>libdwarf</code> object	Initialize the <code>Dwarf_Debug</code> consumer object by calling <code>dwarf_elf_init</code> and using the ELF descriptor.
Notify <code>libddpi</code> object about <code>libdwarf</code> object.	Call <code>ddpi_access_set_debug</code> to let the <code>Ddpi_Info</code> object know about the newly created DWARF consumer object. This is done only once per module/program object.
Use <code>libdwarf</code> and <code>libddpi</code> operations.	<code>libdwarf</code> operations are called to retrieve the DWARF data. For example, <code>dwarf_get_globals</code> will retrieve the list of global symbol entries, and <code>ddpi_module_get_major_name</code> will retrieve the major name from the given <code>Ddpi_Module</code> object.
Terminate the <code>libdwarf</code> object.	<code>dwarf_finish</code> is called to terminate the <code>Dwarf_Debug</code> object.
Terminate ELF descriptor.	The ELF descriptor is terminated with <code>elf_end</code> .
Terminate the <code>libddpi</code> objects.	<code>ddpi_finish</code> is called to terminate the <code>Ddpi_Info</code> object.

Accessing debugging information

This information discusses how to set up access to the debugging information in the executable module. There are two possible types of debugging information:

- Debugging information generated by IBM z/OS XL C and C++ compilers
- Debugging information generated by another compiler

The `libddpi` operations have been created specifically to set up access to executable modules created with the z/OS XL C/C++ compilers. Accordingly, most of this information discusses how to use these functions.

Note: If a module has been created with another compiler, more development must be done to take the place of these operations. For more information, see “Accessing non-z/OS XL C/C++ debugging information” on page 20.

Accessing z/OS XL C/C++ debugging information

This section applies to all modules/program objects that have been compiled with the z/OS XL C and C++ compilers. These modules contain information that allows `libddpi` operations to gain access to the relevant debugging information.

The `ddpi_module_extract_C_CPP_information` operation can determine if the executable module is made up of z/OS XL C/C++ compilation units (CUs). If so, the operation:

- Identifies all the C/C++ CUs within the module and creates a `Ddpi_Elf` object for each CU.
- Locates the ELF object file for each CU.

The recommended method for creating DWARF debugging information for a module/program object is by compiling it with the z/OS XL C/C++ `DEBUG` compiler option. This creates CU objects, each with its own ELF object file. Each CU object contains the name and location of the corresponding ELF object file and an MD5 signature.

Note: For more information about the `DEBUG` option, refer to the *z/OS XL C/C++ User's Guide*.

If a CU object was created with the `DEBUG` compiler option, the `ddpi_elf_get_elf_file_name` operation can retrieve the name and location of the corresponding ELF object file, otherwise, it returns `DW_DLV_NO_ENTRY`.

If the location of the ELF object file cannot be determined, you must provide the location of an ELF object file if it exists, or initialize a conversion process. For more information, see “Accessing and converting ISD information.”

Finally, the addresses within the ELF object file must be relocated to match the loaded executable module. The `ddpi_elf_load_cu` operation:

- Verifies the contents of the ELF object file by making sure that the MD5 signature within the CU object and the ELF object file is the same
- Relocates the ELF object file using the data found within the `.debug_ppa` section

Note: For more information about the using the `.debug_ppa` section for relocations, see “Steps for preparing the `debug_ppa` section” on page 39.

Accessing and converting ISD information

CDA defines consumer functions (operations) that process DWARF debugging information. If the debugging information is in a non-DWARF format, it has to be converted before it can be used by the CDA libraries.

ISD information is created by compiling with the IBM z/OS XL C/C++ compiler with the `TEST` compiler option. Unlike the `DEBUG` compiler option, the `TEST` compiler option does not create an ELF object file. To use ISD information, it must be converted to an ELF object file.

There are two methods that can be used to convert ISD information:

- `isdcnvt` utility

This stand-alone utility extracts ISD information from within CU object files and converts it to the DWARF format in an ELF object file. You can use this to create

all the ELF object files for the CU objects that must be created before you can debug information within the CU objects. Because the location of the ELF object file is not recorded within the CU object file, it is your responsibility to locate the converted ELF object file when accessing debug information in these CU objects.

Note: For more information about the `isdcnvt` utility, see Chapter 6, “In Storage Debug (ISD) Information Conversion Utility,” on page 47.

- `libddpi` conversion operations

The `ddpi_convert_c_cpp_isdobj` and `ddpi_fp_convert_c_cpp_isdobj` operations can be called by any `libddpi` user during run time to convert CU objects containing ISD information into DWARF format. If you are converting a CU object that is part of a loaded executable module, it is not necessary to relocate the resulting ELF object file.

Note: This method affects run-time performance. For more information see *z/OS Common Debug Architecture Library Reference*.

Accessing non-z/OS XL C/C++ debugging information

Extraction is started by calling the `ddpi_module_extract_C_CPP_information` operation. If the executable module was not compiled with the IBM z/OS XL C/C++ compiler, then the format of the debugging information will be unknown to the CDA libraries. You must create your own conversion process in order to use the CDA libraries. That is, you will be responsible for identifying the CUs within the executable module, and adding the necessary information within the `Ddпи_Elf` objects. For more information on how to create a converter application, see Chapter 4, “Using producer APIs,” on page 39.

Chapter 3. Using consumer functions

This information explains how to create a CDA-compliant consumer application that uses the `libelf` and `libdwarf` libraries. It provides an example of the basic structure for a `libdwarf` application that reads ELF object files.

Creating a CDA-compliant consumer application

Note: This information requires that you are familiar with the DWARF format. For more information about DIEs and their structure, see *DWARF/ELF Extensions Library Reference*.

This process is discussed in the following three sections:

- “Initializing `libelf` and `libdwarf`”
- “Consuming DWARF data” on page 36
- “Terminating `libelf` and `libdwarf`” on page 38

Note: ELF object files are created by the `isdcnvt` utility, or by the `DEBUG` option of the z/OS XL C/C++ compiler.

Initializing `libelf` and `libdwarf`

This information describes how the consumer application initializes `libdwarf` with the data to be consumed. Process information may serve as a model for other `libdwarf` consumer applications.

Steps to initialize `libdwarf`

1. Identify the ELF object file containing the data to be used.
2. Create an ELF descriptor to represent the data in the file.
3. Create a `Dwarf_Debug` object to represent the DWARF data contained within the ELF descriptor.

The application uses the `elf_begin` operation to create an ELF descriptor. This operation requires a file descriptor for the ELF object file. For example, the application is given the name of the file from a command line parameter. It then acquires the descriptor with the following code:

```
fd = open(opts.file_name, O_RDONLY);
```

The next step is to create an ELF descriptor with the given ELF object file, using the following code:

```
Elf_Cmd cmd = ELF_C_READ;
Elf *elf;
elf = elf_begin(fd, cmd, NULL);
```

Note: Other operations that can be used are `elf_begin_b` and `elf_begin_c`. Consult the `libelf` documentation for details on using these operations.

The application then determines if the ELF descriptor represents a 32-bit ELF object or a 64-bit ELF object. It uses the `elf32_getehdr` and `elf64_getehdr` operations. For example:

```
Elf32_Ehdr *eh32;
Elf64_Ehdr *eh64;
eh32 = elf32_getehdr(elf);
eh64 = elf64_getehdr(elf);
```

After this sequence the ELF descriptor has been identified as:

- 32 bit if eh32 is not NULL
- 64 bit if eh32 is NULL, and eh64 is not NULL
- Unknown if eh32 and eh64 are both NULL

If the object format is unknown, then the application ignores the object and exits. Otherwise, it continues to create a Dwarf_Debug object. The following code creates a Dwarf_Debug object:

```
Dwarf_Error err;
Dwarf_Debug dbg;
int dres;
dres = dwarf_elf_init(elf, DW_DLC_READ, NULL, NULL, &dbg, &err);
```

It is important to check the return code to ensure that the processing succeeded. dwarf_elf_init returns DW_DLX_OK on successful completion. It returns DW_DLX_ERROR if an error occurs. dwarf_elf_init returns DW_DLX_NO_ENTRY if the ELF descriptor does not contain DWARF data.

If the processing was successful, then dbg contains the Dwarf_Debug object which is used to interact with libdwarf.

Steps to relocate addresses within an ELF file

This information provides code examples that demonstrate how to use libelf operations to relocate addresses within the ELF file.

Before you can run “elfload.c” on page 23, you must provide the reloc_adj array, which is the relocation array that contains adjustments that need to be made to each relocation entry.

1. Compile elfload.c:

```
c89 -qxplink -qlanglvl=extended
-I/usr/lpp/cbclib/include
elfload.c
/usr/lpp/cbclib/lib/libelfdwarf32.x
-o elfload
```

2. Run elfload.o, using the following command: elfload

The return code should be 0. The generated debug file mytest.dbg should have the following .symtab entries:

```
Sect 20 .symtab symtab off=0x2559 0x26a9 size=336 addr=0x0 align=1 flag=0x0 [---] esize=16 info=21 link=19
String table = ".strtab"
Sym 0: value= 0x0000, size= 0 sect= undef, type= none, bind= local, name=
Sym 1: value= 0x0000, size= 0 sect= abs, type= file, bind= local, name= /c390/cbc/zosdev/nightly/libmd5/src/md5.c
Sym 2: value= 0x37b8, size= 0 sect= .text, type= none, bind= local, name= .ppa2_b_3C2C968222FFB7242B5253006501F60F
Sym 3: value= 0x0000, size= 1 sect= .debug_info, type= sect, bind= local, name=
```

Example: Relocating addresses within an ELF file

This example that demonstrates how to use `libelf` operations to relocate addresses within the ELF file.

For the task steps, see “Steps to relocate addresses within an ELF file” on page 22.

In `elfload.c`, the following variables are hardcoded:

- In `main`, the name of the test subject `.dbg` file (`mytest.dbg`).
- In `main`, the MD5 signature found in the `.dbg` file (variable `md5`).
- In `relocate_elf_load_cu()`, the target relocation address (variable `reloc_adj`) contains the address delta to be applied to target address.

`elfload.c`

```
#include <stdlib.h>
#include "libelf/libelf.h"

/*****
 Structure used to keep track of information within ELF
 *****/
typedef unsigned long int uint64;
typedef signed long long int int64;
typedef char bool;

/* ELF symbol details
*/
typedef struct ElfSymbol_s {
    char* es_name; /* ELF symbol: name */
    uint64 es_value; /* value */
    uint64 es_size; /* size */
    unsigned char es_type; /* type */
    unsigned char es_bind; /* bind */
    unsigned char es_other; /* other */
    int64 es_shndx; /* ELF section index */
} *ElfSymbol;

/* ELF file details
*/
typedef struct ElfDetails_s {
    Elf* ed_elf; /* ->ELF instance for CU */
    bool ed_is_64bit; /* 64-bit: true */
    /* 32-bit: false */

    /* ELF Section details */
    Elf_Scn** ed_elf_scns; /* List of ->ELF scn objects */
    char** ed_scn_names; /* List of ELF section names */
    int64* ed_infos; /* List of section sh_info values */
    char** ed_datas; /* List of ->section data buffer */
    uint64* ed_data_sizes; /* List of length of section data */
    int64 ed_n_elf_scns; /* Number of ELF sections */

    int64 ed_text_idx; /* .text section index */
    int64 ed_syntab_idx; /* .syntab section index */
    int64 ed_strtab_idx; /* .strtab section index */
    int64 ed_shstrtab_idx; /* .shstrtab section index */

    /* ELF Symbol details */
    ElfSymbol ed_symbols; /* List of ->ELF symbol info */
    uint64 ed_n_symbols; /* Number of ELF symbols */
} *ElfDetails;

/*--< Local Routines >-----*/
```

```

| /* Examine ELF descriptor and find out all information necessary
|   for relocating the .dbg.
|   All information are stored in 'ret_details'.
|   'ret_details' is deallocated with _load_elf_term
| */
| static int
|   _load_elf_file_details(
|     Elf*          elf,          /* ->ELF instance for CU      I*/
|     ElfDetails*  ret_details); /* ->returned ELF file details  0*/
|
| /* Terminate ELF loader processing, release resources
| */
| static int
|   _load_elf_term(
|     ElfDetails  details);      /* ELF file details          I*/
|
| /* Load 64-bit ELF symbol table
| */
| static int
|   _load_elf64_symbol_table(
|     ElfDetails  details);      /* ELF file details          I*/
|
| /* Load 32-bit ELF symbol table
| */
| static int
|   _load_elf32_symbol_table(
|     ElfDetails  details);      /* ELF file details          I*/
|
| /* Given the 16 byte raw MD5 signature, verify that it matches the loaded
|   .dbg file
| */
| static int
|   _validate_MD5_signature(
|     ElfDetails  details,        /* ELF file details          I*/
|     unsigned char digest[16]); /* PPA2 MD5 signature        I*/
|
| /* Relocate the ELF sections based on the information in 'reloc_adj'
| */
| static int
|   _relocate_elf_sections(
|     ElfDetails  details,        /* ELF file details          I*/
|     int64*      reloc_adj);     /* Adjustment array         I*/
|
| /*--< Relocation main routines >-----*/
| int
|   relocate_elf_load_cu(
|     Elf*          elf,          /* ->ELF instance for CU      I0*/
|     unsigned char md5_sig[16]); /* MD5 signature              I*/
|
| /*--< FUNCTION IMPLEMENTATION >-----*/
| int main ()
| {
|   Elf* elf;
|   FILE *fp;
|   int rc;
|   unsigned char md5[16] = { 0x3C, 0x2C, 0x96, 0x82,
|                             0x22, 0xFF, 0xB7, 0x24,
|                             0x2B, 0x52, 0x53, 0x00,
|                             0x65, 0x01, 0xF6, 0x0F };
|
|   elf_version (EV_CURRENT);
|
|   fp = fopen ("mytest.dbg", "rb");
|   elf = elf_begin_b (fp, ELF_C_READ, NULL);
|
|   rc = relocate_elf_load_cu (elf, md5);
|   printf ("rc should be zero: %d\n", rc);

```

```

|     elf_end(elf);
| }
|
| #pragma convert ("ISO8859-1")
| /* Load ELF file and relocate .text to given address(es)
| */
| int
| relocate_elf_load_cu(
|     Elf*           elf,           /* ->ELF instance for CU       IO*/
|     unsigned char md5_sig[16]) { /* MD5 signature             I*/
| ElfDetails
|     details;
| int64*
|     reloc_adj;           /* An array keeping track of
|                          address adjustment needed for
|                          each .text symid          */
|
|     int i;
|     int rc;
|
|     /* Load ELF file section and symbol tables */
|     rc = _load_elf_file_details(elf, &details);
|     if (rc) return rc;
|
|     /* Validate MD5 signature */
|     rc = _validate_MD5_signature(details, md5_sig);
|     if (rc) return rc;
|
|     /* TO BE FILLED IN: create reloc_adj array */
|     /* This will relocate 0x37b8 to 0xDEADBEEF */
|     reloc_adj = (int64*) calloc (sizeof(int64), details->ed_n_symbols);
|     reloc_adj[2] = 0xDEADBEEF;
|
|     /* Relocate the ELF sections based on the current section origins */
|     rc = _relocate_elf_sections(details, reloc_adj);
|     if (rc) return rc;
|
|     /* Processing complete. Remove temporary tables */
|     rc = _load_elf_term(details);
|     if (rc) return rc;
|
|     /* Terminate reloc_adj */
|     free (reloc_adj);
|
|     return 0;
| }
|
| /* Load ELF file section and symbol tables
| */
| static int
| _load_elf_file_details(
|     Elf*           elf,           /* ->ELF instance for CU       I*/
|     ElfDetails*   ret_details) { /* ->returned ELF file details  O*/
|
|     ElfDetails
|         details;
|
|     char*
|         ehdr_ident;
|
|     Elf32_Ehdr*
|         ehdr32;
|     Elf64_Ehdr*
|         ehdr64;
|
|     Elf32_Shdr*
|         shdr32;
|     Elf64_Shdr*
|         shdr64;
|
|     Elf_Scn*
|         scn;
|
|     Elf_Data*
|         data;
|
|     char*
|         scn_name;

```

```

| Elf_Scn**      section_list;
| char**        name_list;
| char**        data_list;
| uint64*       data_size_list;
| int64*        info_list;
|
| int64         scn_idx,
|               n_elf_scns,
|               shstrtab_idx;
|
| int           rc,
|               is_64bit,
|               elf_machine;
|
| /* Determine if 64-bit or 32-bit ELF file */
| if ((ehdr_ident = elf_getident(elf, NULL)) == NULL) {
|     return -1; /* ERROR */
| }
| is_64bit = (ehdr_ident[EI_CLASS] == ELFCLASS64);
|
| /* Access the ELF file header */
| if (is_64bit) {
|     if ((ehdr64 = elf64_getehdr(elf)) == NULL) {
|         return -1; /* ERROR */
|     }
|     elf_machine = ehdr64->e_machine;
|     n_elf_scns = ehdr64->e_shnum + 1; /* Allow for section 0 */
|     shstrtab_idx = ehdr64->e_shstrndx;
| }
| else {
|     if ((ehdr32 = elf32_getehdr(elf)) == NULL) {
|         return -1; /* ERROR */
|     }
|     elf_machine = ehdr32->e_machine;
|     n_elf_scns = ehdr32->e_shnum + 1; /* Allow for section 0 */
|     shstrtab_idx = ehdr32->e_shstrndx;
| }
|
| /* Validate machine type */
| if (elf_machine != EM_S390) {
|     return -1; /* ERROR */
| }
|
| /* Allocate the new ElfDetails object */
| if (n_elf_scns == 0) {
|     return -1; /* ERROR */
| }
|
| details = (ElfDetails) calloc (sizeof(struct ElfDetails_s), 1);
| if (details == NULL) {
|     return -2; /* out of memory */
| }
|
| /* Initialize the new object */
| details->ed_elf = elf;
| details->ed_n_elf_scns = n_elf_scns;
| details->ed_shstrtab_idx = shstrtab_idx;
|
| if (is_64bit) {
|     details->ed_is_64bit = 1;
| }
|
| /* Allocate list object (array of Dwarf_Ptr) for the ELF sections */
| section_list = (Elf_Scn**) calloc (sizeof(Elf_Scn*), n_elf_scns);
| if (section_list == NULL) {
|     return -2; /* out of memory */
| }

```

```

| }
| details->ed_elf_scns = section_list;
|
| /* Allocate list object (array of char*) for the ELF section names */
| name_list = (char**) calloc (sizeof(char*), n_elf_scns);
| if (name_list == NULL) {
|     return -2; /* out of memory */
| }
| details->ed_scn_names = name_list;
|
| /* Allocate list object (array of Dwarf_Ptr) for section data addrs */
| data_list = (char**) calloc (sizeof(char*), n_elf_scns);
| if (data_list == NULL) {
|     return -2; /* out of memory */
| }
| details->ed_datas = data_list;
|
| /* Allocate addr object (array of uint64) for section data lengths */
| data_size_list = (uint64*) calloc (sizeof(uint64), n_elf_scns);
| if (data_size_list == NULL) {
|     return -2; /* out of memory */
| }
| details->ed_data_sizes = data_size_list;
|
| /* Allocate addr object (array of int64) for section sh_info */
| info_list = (int64*) calloc (sizeof(int64), n_elf_scns);
| if (info_list == NULL) {
|     return -2; /* out of memory */
| }
| details->ed_infos = info_list;
|
| /* Populate the ELF section lists */
| scn_idx = 0;
| scn = NULL;
| while ((scn = elf_nextscn(elf,scn)) != NULL) {
|
|     /* Save ELF section for section symbol lookup */
|     scn_idx = elf_ndxscn(scn);
|     if (scn_idx < n_elf_scns) {
|         section_list[scn_idx] = scn;
|     }
|     else {
|         return -1; /* ERROR */
|     }
|
|     /* Process ELF section header */
|     if (is_64bit) {
|         if ((shdr64 = elf64_getshdr(scn)) == NULL) {
|             return -1; /* ERROR */
|         }
|
|         /* Get section name */
|         if ((scn_name = elf_strptr(elf,
|                                   shstrtab_idx,
|                                   shdr64->sh_name)) == NULL) {
|             return -1; /* ERROR */
|         }
|
|         info_list[scn_idx] = shdr64->sh_info;
|     }
|     else {
|         if ((shdr32 = elf32_getshdr(scn)) == NULL) {
|             return -1; /* ERROR */
|         }
|
|         /* Get section name */
|         if ((scn_name = elf_strptr(elf,

```

```

|                                     shstrtab_idx,
|                                     shdr32->sh_name)) == NULL) {
|     return -1; /* ERROR */
| }
|
| info_list[scn_idx] = shdr32->sh_info;
| }
|
| /* Note ELF Section names */
| name_list[scn_idx] = scn_name;
|
| /* Note index of ELF .text, .symtab, .strtab and .shstrtab sections */
| if (strcmp(scn_name, ".text") == 0) {
|     /* Validate .text is z/OS DWARF in ELF packing */
|     if (is_64bit) {
|         if (shdr64->sh_type != SHT_NOBITS) {
|             return -1; /* ERROR */
|         }
|     }
|     else {
|         if (shdr32->sh_type != SHT_NOBITS) {
|             return -1; /* ERROR */
|         }
|     }
|
|     /* Validate there is only 1 .text section */
|     if (details->ed_text_idx != 0) {
|         return -1; /* ERROR */
|     }
|     details->ed_text_idx = scn_idx;
| }
|
| else if (strcmp(scn_name, ".symtab") == 0) {
|     /* Validate there is only 1 .symtab section */
|     if (details->ed_symtab_idx != 0) {
|         return -1; /* ERROR */
|     }
|     details->ed_symtab_idx = scn_idx;
| }
|
| else if (strcmp(scn_name, ".strtab") == 0) {
|     /* Validate there is only 1 .strtab section */
|     if (details->ed_strtab_idx != 0) {
|         return -1; /* ERROR */
|     }
|     details->ed_strtab_idx = scn_idx;
| }
|
| else if (strcmp(scn_name, ".shstrtab") == 0) {
|     /* Validate there is only 1 .shstrtab section */
|     if (details->ed_shstrtab_idx != scn_idx) {
|         return -1; /* ERROR */
|     }
| }
|
| /* Prepare to read ELF section Data */
| if ((data = elf_getdata(scn, 0)) != NULL) {
|     data_list[scn_idx] = data->d_buf;
|     data_size_list[scn_idx] = data->d_size;
| }
| }
|
| /* Ensure the file has all required sections */
| if ((details->ed_text_idx == 0) ||
|     (details->ed_symtab_idx == 0) ||
|     (details->ed_strtab_idx == 0) ||
|     (details->ed_shstrtab_idx == 0)) {

```



```

|     return -1; /* ERROR */
| }
|
| /* Create the symbol table from the ELF .symtab section */
| if (details->ed_is_64bit) {
|     rc = _load_elf64_symbol_table(details);
| }
| else {
|     rc = _load_elf32_symbol_table(details);
| }
| if (rc) return rc;
|
| /* Return the ElfDetails object to the caller */
| *ret_details = details;
|
| return 0;
| }
|
| /* Terminate ELF loader processing, release resources
| */
| static int
| _load_elf_term(
|     ElfDetails      details) {          /* ELF file details          I*/
|
|     /* Delete the resources for this ElfDetails object */
|     if (details->ed_elf_scns != NULL) {
|         free (details->ed_elf_scns);
|     }
|
|     if (details->ed_datas != NULL) {
|         free (details->ed_datas);
|     }
|
|     if (details->ed_data_sizes != NULL) {
|         free (details->ed_data_sizes);
|     }
|
|     if (details->ed_symbols != NULL) {
|         free (details->ed_symbols);
|     }
|
|     free (details);
|
|     return 0;
| }
|
| /* Load 64-bit ELF symbol table
| */
| static int
| _load_elf64_symbol_table(
|     ElfDetails      details) {          /* ELF file details          I*/
|     Elf*            elf;
|
|     Elf64_Shdr*     shdr64;
|     Elf64_Sym*      symtab;
|
|     ElfSymbol       symbols,
|                    cur_sym;
|
|     uint64          link,
|                    shstrtab_idx;
|
|     uint64          n_symbols,
|                    i;
|
|     elf = details->ed_elf;
|     if (elf == NULL) {

```

```

    return -1; /* ERROR */
}

shstrtab_idx = details->ed_shstrtab_idx;

/* Allocate the array of ElfSymbol objects */
n_symbols = (details->ed_data_sizes[details->ed_syntab_idx]) /
            sizeof(Elf64_Sym);
if (n_symbols == 0) {
    return -1; /* ERROR */
}
symbols = (ElfSymbol) calloc (sizeof(struct ElfSymbol_s), n_symbols);
if (symbols == NULL) {
    return -2; /* Out of memory */
}
details->ed_symbols = symbols;
details->ed_n_symbols = n_symbols;

/* Process the 64-bit .syntab section */
cur_sym = symbols;
syntab = (Elf64_Sym*)(details->ed_datas[details->ed_syntab_idx]);
link = details->ed_strtab_idx;
for (i = 0;
     i < n_symbols;
     i++, cur_sym++, syntab++) {
    cur_sym->es_value = syntab->st_value;
    cur_sym->es_size = syntab->st_size;
    cur_sym->es_type = ELF64_ST_TYPE(syntab->st_info);
    cur_sym->es_bind = ELF64_ST_BIND(syntab->st_info);
    cur_sym->es_other = syntab->st_other;
    cur_sym->es_shndx = syntab->st_shndx;

    if (syntab->st_name == 0) {
        if (cur_sym->es_type == STT_SECTION) {
            if (cur_sym->es_shndx == SHN_UNDEF) {
                cur_sym->es_name = "undef";
            }
            else if (cur_sym->es_shndx == SHN_ABS) {
                cur_sym->es_name = "abs";
            }
            else if (cur_sym->es_shndx == SHN_COMMON) {
                cur_sym->es_name = "common";
            }
            else if (cur_sym->es_shndx < details->ed_n_elf_scns) {
                /* Get ELF section header */
                shdr64 = elf64_getshdr(details->ed_elf_scns[cur_sym->es_shndx]);
                if (shdr64 == NULL) {
                    return -1; /* ERROR */
                }

                /* Get ELF section name */
                cur_sym->es_name = elf_strptr(details->ed_elf,
                                             shstrtab_idx,
                                             shdr64->sh_name);
            }
            else {
                cur_sym->es_name = "<Unknown section=">";
            }
        }
        else {
            /* Not section... note NULL */
            cur_sym->es_name = "<NULL>";
        }
    }
    else {
        cur_sym->es_name = elf_strptr(details->ed_elf,
                                     link,

```

```

|                                     symtab->st_name);
|     }
|     if (cur_sym->es_name == NULL) {
|         return -1; /* ERROR */
|     }
| }
|
| return 0;
| }
|
| /* Load 32-bit ELF symbol table
| */
| static int
| _load_elf32_symbol_table(
|     ElfDetails details) {          /* ELF file details          I*/
|     Elf* elf;
|
|     Elf32_Shdr* shdr32;
|     Elf32_Sym* symtab;
|
|     ElfSymbol symbols,
|         cur_sym;
|
|     uint64 link,
|         shstrtab_idx;
|
|     uint64 n_symbols,
|         i;
|
|     elf = details->ed_elf;
|     if (elf == NULL) {
|         return -1; /* ERROR */
|     }
|
|     shstrtab_idx = details->ed_shstrtab_idx;
|
|     /* Allocate the array of ElfSymbol objects */
|     n_symbols = (details->ed_data_sizes[details->ed_symtab_idx] /
|                 sizeof(Elf32_Sym));
|
|     if (n_symbols == 0) {
|         return -1; /* ERROR */
|     }
|
|     symbols = (ElfSymbol) calloc (sizeof(struct ElfSymbol_s), n_symbols);
|     if (symbols == NULL) {
|         return -2; /* Out of memory */
|     }
|     details->ed_symbols = symbols;
|     details->ed_n_symbols = n_symbols;
|
|     /* Process the 32-bit .symtab section */
|     cur_sym = symbols;
|     symtab = (Elf32_Sym*)(details->ed_datas[details->ed_symtab_idx]);
|     link = details->ed_strtab_idx;
|     for (i = 0;
|         i < n_symbols;
|         i++, cur_sym++, symtab++) {
|         cur_sym->es_value = symtab->st_value;
|         cur_sym->es_size = symtab->st_size;
|         cur_sym->es_type = ELF32_ST_TYPE(symtab->st_info);
|         cur_sym->es_bind = ELF32_ST_BIND(symtab->st_info);
|         cur_sym->es_other = symtab->st_other;
|         cur_sym->es_shndx = symtab->st_shndx;
|
|         if (symtab->st_name == 0) {
|             if (cur_sym->es_type == STT_SECTION) {

```

```

|         if (cur_sym->es_shndx == SHN_UNDEF) {
|             cur_sym->es_name = "undef";
|         }
|         else if (cur_sym->es_shndx == SHN_ABS) {
|             cur_sym->es_name = "abs";
|         }
|         else if (cur_sym->es_shndx == SHN_COMMON) {
|             cur_sym->es_name = "common";
|         }
|         else if (cur_sym->es_shndx < details->ed_n_elf_scns) {
|             /* Get ELF section header */
|             shdr32 = elf32_getshdr(details->ed_elf_scns[cur_sym->es_shndx]);
|             if (shdr32 == NULL) {
|                 return -1; /* ERROR */
|             }
|
|             /* Get ELF section name */
|             cur_sym->es_name = elf_strptr(details->ed_elf,
|                                         shstrtab_idx,
|                                         shdr32->sh_name);
|
|         }
|         else {
|             cur_sym->es_name = "<Unknown section="">";
|         }
|     }
|     else {
|         /* Not section... note NULL */
|         cur_sym->es_name = "<NULL>";
|     }
| }
| else {
|     cur_sym->es_name = elf_strptr(details->ed_elf,
|                                 link,
|                                 symtab->st_name);
| }
|
| if (cur_sym->es_name == NULL) {
|     return -1; /* ERROR */
| }
| }
|
| return 0;
| }
|
| /* Given the 16 byte raw MD5 signature, verify that it matches the loaded
| .dbg file
| */
| static int
| _validate_MD5_signature(
|     ElfDetails      details,          /* ELF file details          I*/
|     unsigned char   digest[16] {     /* PPA2 MD5 signature       I*/
| ElfSymbol          symbols,
|                   cur_sym;
|
|     unsigned char   md5_chars[32+1];
|     unsigned char*   sym_name;
|
|     uint64          n_symbols,
|                   i,
|                   pos;
|
|     symbols = details->ed_symbols;
|     n_symbols = details->ed_n_symbols;
|     if ((symbols == NULL) ||
|         (n_symbols == 0)) {
|         return -1; /* ERROR */
|     }
| }

```

```

|
| /* Generate text for MD5 signature portion of symbol */
| for (i = 0, pos = 0; i < 16; i++) {
|     const char * convstring = "0123456789ABCDEF";
|     char
|         top_nibble,
|         bottom_nibble;
|
|     top_nibble    = digest[i] >> 4;
|     bottom_nibble = digest[i] & 0x0F;
|     md5_chars[pos] = convstring[top_nibble];
|     pos++;
|     md5_chars[pos] = convstring[bottom_nibble];
|     pos++;
| }
| md5_chars[pos] = 0x00;
|
| /* Scan the symbol table for the first symbol in .text that resemble MD5 signature */
| for (i = 0, cur_sym = symbols;
|     i < n_symbols;
|     i++, cur_sym++) {
|     const int sym_name_len = strlen(cur_sym->es_name);
|     sym_name = cur_sym->es_name;
|     if (cur_sym->es_shndx == details->ed_text_idx &&
|         sym_name_len >= 32 &&
|         !strcmp(sym_name+sym_name_len-32, md5_chars)) {
|         /* matching MD5 signature found */
|         return 0;
|     }
| }
|
| /* MD5 signature not found */
| return -1;
| }
|
| /* Relocate the ELF sections based on the relocation adjustments array
| 'reloc_adj' is an array containing adjustments that needs to be
| made to each corresponding relocation entry.
| For example:
| Typical .symtab entries:
| Sym 2: value= 0x000, ..., name= .MD5_3FD489E1D88CB743682E3A44875A1765
| Sym 3: value= 0x010, ..., name= func1
| Sym 4: value= 0x020, ..., name= func2
| Sym 5: value= 0x050, ..., name= func3
| If all relocation base on sym 2, and it needs to adjust to 0xDEADBEEF, then
| 'reloc_adj' would contain:
| { 0, 0, 0xDEADBEEF, 0, 0, 0 }
|     ^-- index 2 correspond to sym 2
| */
| static int
| _relocate_elf_sections(
|     ElfDetails      details,          /* ELF file details          I*/
|     int64*          reloc_adj) {      /* .text relocation adjustments I*/
|     ElfSymbol
|         symbols,
|         cur_sym;
|
|     uint64          reloc_offset;
|     uint64          reloc_sym;
|     int64           reloc_scn;
|
|     char**          scn_names;
|     int64*          infos;
|
|     unsigned int    reloc_type;
|
|     char*           scn_name,
|     *               relscn_name,
|     *               sym_name,

```

```

|      *          reloc_scn_name,
|      *          reloc_name;
|
| char**          datas;
| uint64*        data_sizes;
|
| int64          relscn_idx;
|
| char*          reloc_data;
| char*          relscn_data;
| uint64         reloc_data_size,
|               reloc_data_off,
|               relscn_data_size;
|
| int64          n_elf_scns,
|               change;
|
| uint64         n_symbols,
|               i;
|
| n_elf_scns    = details->ed_n_elf_scns;
| n_symbols     = details->ed_n_symbols;
| scn_names     = details->ed_scn_names;
| symbols       = details->ed_symbols;
| datas        = details->ed_datas;
| data_sizes    = details->ed_data_sizes;
| infos        = details->ed_infos;
|
| if ((n_symbols == 0) ||
|     (n_elf_scns == 0) ||
|     (scn_names == NULL) ||
|     (symbols == NULL) ||
|     (datas == NULL) ||
|     (data_sizes == NULL)) {
|     return -1; /* ERROR */
| }
|
| /* Scan section lists, processing SHT_REL-format relocation sections */
| for (i = 1;
|      i < n_elf_scns;
|      i++) {
|
|     /* Check for ELF SHT_REL-format section */
|     scn_name = scn_names[i];
|     if (strncmp(scn_name, ".rel.",5) == 0) {
|
|         /* Access relocation section info */
|         reloc_data = datas[i];
|         reloc_data_size = data_sizes[i];
|
|         /* Access data section info */
|         relscn_idx = infos[i];
|         relscn_name = scn_names[relscn_idx];
|         relscn_data = datas[relscn_idx];
|         relscn_data_size = data_sizes[relscn_idx];
|
|         if (details->ed_is_64bit) {
|             /* Relocate all R_390_64 type relocation entries */
|             for (reloc_data_off = 0;
|                  reloc_data_off < reloc_data_size;
|                  reloc_data_off += sizeof(Elf64_Rel)) {
|                 Elf64_Rel* p = (Elf64_Rel*)(reloc_data + reloc_data_off);
|
|                 reloc_offset = p->r_offset;
|
|                 reloc_sym = ELF64_R_SYM(p->r_info);
|                 if (reloc_sym >= n_symbols) {

```

```

    return -1; /* ERROR */
}

cur_sym = symbols + reloc_sym;
reloc_scn = cur_sym->es_shndx;
if (reloc_scn >= n_elf_scns) {
    return -1; /* ERROR */
}

reloc_type = ELF64_R_TYPE(p->r_info);
switch (reloc_type) {

    case R_390_NONE :
        /* No adjustment required... likely DWARF info */
        break;

    case R_390_32 : {
        /* Check for relocation adjustment */
        signed int* relscn_ptr;
        signed int  reloc_item;

        change = reloc_adj[reloc_sym];
        if (change != 0) {
            relscn_ptr = (signed int*)(relscn_data + reloc_offset);
            reloc_item = *relscn_ptr;
            *relscn_ptr = reloc_item + change;
        }
    }
    break;

    case R_390_64 : {
        /* Check for relocation adjustment */
        int64* relscn_ptr;
        int64  reloc_item;

        change = reloc_adj[reloc_sym];
        if (change != 0) {
            relscn_ptr = (int64*)(relscn_data + reloc_offset);
            reloc_item = *relscn_ptr;
            *relscn_ptr = reloc_item + change;
        }
    }
    break;

    default :
        return -1; /* ERROR */
}
}

else {
    /* Relocate all R_390_32 type relocation entries */
    for (reloc_data_off = 0;
         reloc_data_off < reloc_data_size;
         reloc_data_off += sizeof(Elf32_Rel)) {
        Elf32_Rel* p = (Elf32_Rel*)(reloc_data + reloc_data_off);

        reloc_offset = p->r_offset;

        reloc_sym = ELF32_R_SYM(p->r_info);
        if (reloc_sym >= n_symbols) {
            return -1; /* ERROR */
        }

        cur_sym = symbols + reloc_sym;
        fflush(NULL);
        reloc_scn = cur_sym->es_shndx;
    }
}

```

```

|         if (reloc_scn >= n_elf_scns) {
|             return -1; /* ERROR */
|         }
|
|         reloc_type = ELF32_R_TYPE(p->r_info);
|         switch (reloc_type) {
|
|             case R_390_NONE :
|                 /* No adjustment required... likely DWARF info */
|                 break;
|
|             case R_390_32 : {
|                 /* Check for relocation adjustment */
|                 signed int*   relscn_ptr;
|                 signed int    reloc_item;
|
|                 change = reloc_adj[reloc_sym];
|                 if (change != 0) {
|                     relscn_ptr = (signed int*)(relscn_data + reloc_offset);
|                     reloc_item = *relscn_ptr;
|                     *relscn_ptr = reloc_item + change;
|                 }
|             }
|             break;
|
|             case R_390_64 : {
|                 /* Check for relocation adjustment */
|                 int64* relscn_ptr;
|                 int64  reloc_item;
|
|                 change = reloc_adj[reloc_sym];
|                 if (change != 0) {
|                     relscn_ptr = (int64*)(relscn_data + reloc_offset);
|                     reloc_item = *relscn_ptr;
|                     *relscn_ptr = reloc_item + change;
|                 }
|             }
|             break;
|
|             default :
|                 return -1; /* ERROR */
|         }
|     }
| }
|
| }
|
|     return 0;
| }
| #pragma convert (0)

```

Consuming DWARF data

Once a Dwarf_Debug object has been created, its data may be used by the program analysis application. This information discusses how the application uses libdwarf operations to extract information from its DWARF objects. That is, it describes how a consumer function in the application can:

- Traverse the Debug Information Entry (DIE) hierarchy.
- Access information contained in DIEs.

Traversing the DIE hierarchy

This information describes how a program analysis application traverses the DIE hierarchy in the `.debug_ppa` section. The steps are the same for any function that traverses any DWARF DIE section.

The first step is to obtain a `Dwarf_Section` object representing the `.debug_ppa` section. For example:

```
dwarf_debug_section(dbg,
                    DW_SECTION_DEBUG_PPA,
                    DW_SECTION_IS_DEBUG_DATA,
                    &section, &err);
```

Now that the application has the `.debug_ppa` section, it will step through all the unit headers with the following code:

```
/* Loop until it returns 0 */
unit_offset = 0;
while( (nres = dwarf_next_unit_header(dbg,
                                     section,
                                     &unit_header_length,
                                     &version_stamp,
                                     &abbrev_offset,
                                     &address_size,
                                     &next_unit_offset,
                                     &err)
) == DW_DLV_OK ) {
    /* Process this unit header. */
    unit_offset = next_unit_offset;
}
```

For each iteration of the above loop, the application obtains the root DIE of that unit by using the following call:

```
dwarf_rootof(section, unit_offset, &root_die, &err);
```

Once the application has the root DIE, it can traverse all children of the root DIE by using the `dwarf_child` operation as follows:

```
dwarf_child(in_die, &child, &err);
```

The `in_die` variable is the root DIE. The program analysis application continues processing children until the above `dwarf_child` operation returns `DW_DLV_NOENTRY` (indicating that it has reached the bottom of the hierarchy).

The program analysis application now proceeds to traverse the siblings of the root DIE by using the `dwarf_siblingof` operation. For example:

```
dwarf_siblingof(dbg, in_die, &sibling, &err);
```

Accessing information in a DIE

This information lists the `libdwarf` operations used by application to access data within a DIE.

Table 4. DIE access operations

Call	Description
<pre>dwarf_tag(die, &tag, &err);</pre>	This call retrieves the TAG of a DIE.

Table 4. DIE access operations (continued)

Call	Description
<code>dwarf_diename(dbg, &tagname, &err);</code>	This call retrieves the name of a TAG.
<code>dwarf_dieoffset(die, &overall_offset, &err);</code>	This call retrieves the overall offset of a DIE.
<code>dwarf_die_CU_offset(die, &offset, &err);</code>	This call retrieves the offset of a DIE within a given compilation unit.
<code>dwarf_attrlist(die, &atlist, &atcnt, &err);</code>	This call retrieves a list of the attributes for a DIE.
<code>dwarf_formudata(attrib, &val, &err);</code>	This call retrieves the unsigned value of a given attribute.
<code>dwarf_whatform(attrib, &theform, &err);</code>	This call retrieves the form of a given attribute.

Terminating libelf and libdwarf

This information discusses how the program analysis application terminates its interaction with `libdwarf` and the associated ELF descriptors. The termination order is the opposite of the initialization order; first the `Dwarf_Debug` object is terminated, then the ELF descriptor.

The program analysis application terminates the `Dwarf_Debug` object with the following code:

```
dwarf_finish(dbg, &err);
```

When the `Dwarf_Debug` object has been terminated, the program analysis application terminates the ELF descriptor with the following code:

```
elf_end(elf);
```

Chapter 4. Using producer APIs

This information explains how to create a producer application that converts ISD debugging information into the DWARF format. In this case, only the `libelf` and `libdwarf` libraries are used. The process is similar to that used by the `isdconv` utility, which uses `libddpi` conversion operations.

Creating a producer application

Note: This information requires that you are familiar with the DWARF format. For more information about DIEs and their structure, see *DWARF/ELF Extensions Library Reference*.

The discussion is divided into the following topics:

- “Steps for creating a line-number table”
- “Steps for preparing the `debug_ppa` section”
- “Steps for converting application symbols” on page 40

Finally, an overview example is presented at the end of this information.

Steps for creating a line-number table

Before you begin: Create a CU DIE to hold the ISD information.

Complete the following steps to create a line-number table.

1. Create a `.debug_line` section by calling `dwarf_add_section_to_debug`.
2. For each of the PPA1s, you must choose between a number of options. These options are:
 - a. Either call `dwarf_global_linetable` to add line number information to the CU DIE, or call `dwarf_subprogram_linetable` to add line number information to the subprogram DIE.
 - b. Call `dwarf_line_set_address` to set the relative address at the beginning of the block of lines.
 - c. Call `dwarf_add_line_entry` or `dwarf_add_line_entry_b` for each of the line-number entries.
 - d. Call `dwarf_line_end_sequence` to set the address at the end of the block of lines.

The CU DIE is complete, and you are ready to prepare the `.debug_ppa` section.

Steps for preparing the `debug_ppa` section

This information assumes that:

- Each CU has a corresponding ELF object file.
- The ELF object file contains the addresses of all object information elements within the CU.

When the CU is loaded into a memory image, the addresses of its object information elements are converted to available memory addresses. When the ELF object file is loaded, the addresses that it contains must be properly relocated to

match the new memory addresses of the CU information. The information in the PPA1 and PPA2 blocks of the object file are the key addresses that can be used to relocate the ELF object file.

Complete the following steps to record the address of the PPA1 and PPA2 blocks in the CU and create an MD5 signature for the ELF object.

1. Create a `.debug_ppa` section by calling `dwarf_add_section_to_debug`.
2. Create a PPA2 DIE and add it to the `.debug_ppa` section by calling `dwarf_add_die_to_debug_section`.
3. Create an MD5 signature and add it to the PPA2 DIE by calling `dwarf_add_AT_name`.

The `.debug_ppa` section is complete, and you are ready to convert the symbols.

Steps for converting application symbols

Converting symbols means creating DIEs for the symbol and for the type of symbol. The parent of a global symbol is the CU DIE. The parent of a local symbol is the DIE for the block containing this symbol.

To convert a symbol:

1. Create a DIE by calling the `dwarf_new_die` operation. If the DIE is initially created with a NULL parent, it can be linked later by calling the `dwarf_die_link` operation.
2. Add the applicable attributes to the DIE. Attributes are added by calling the following operations:
 - `dwarf_add_AT_reference` adds a reference to another DIE, such as the type DIE.
 - `dwarf_add_AT_flag` adds a true or false attribute.
 - `dwarf_add_AT_targ_address` adds an address attribute.
 - `dwarf_add_AT_location_expr` adds a location-expression attribute. For more information, see “Location expressions” on page 41.
 - `dwarf_add_AT_name` adds a name to the DIE.
 - `dwarf_add_AT_unsigned_const` adds an unsigned constant as an attribute.
 - `dwarf_add_AT_reference_with_reloc` adds a reference to a CU DIE, so that relocation entries are created.

All of the information about the symbol has been added to DIEs, and the DIEs have been linked. The producer application is complete.

Additional steps

The following lists additional steps for certain types of symbols:

- If the symbol is a function, then create a PPA1 DIE and make it a child of the PPA2 DIE.
- If the symbol is a non-static global variable or function, then add it to the `.debug_pubnames` section by calling `dwarf_add_pubname`.
- If the symbol is a static variable, then add it to the `.debug_varnames` section by calling `dwarf_add_varname`.
- If the symbol is a static function, then add it to the `.debug_funcnames` section by calling `dwarf_add_funcname`.

Location expressions

Location expressions are created by calling the `dwarf_new_expr` operation. Functions are added to the location expression by calling the following operations:

- `dwarf_add_expr_gen` is a generic operation that adds an operator and possibly some operands.
- `dwarf_add_expr_addr` adds the `DW_OP_addr` operator and an address.
- `dwarf_add_expr_reg` adds the `DW_OP_reg` operator for the given register number.
- `dwarf_add_expr_breg` adds the `DW_OP_breg` operator for the given register number.

Example: Converting symbol attributes into DIEs

Before you begin: You need to have created a CU DIE.

To convert a global variable integer (`myvar`), use the following process:

1. Create an `int` DIE if it has not already been created.
 - a. Call `dwarf_new_die`. The tag is `DW_TAG_base_type`, the parent is the CU DIE, the child is `NULL`, and the left and right siblings are `NULL`.
 - b. Call `dwarf_add_AT_name` for the type DIE, giving it the name `int`.
 - c. Call `dwarf_add_AT_unsigned_const` for the type DIE to add the encoding. The attribute should be `DW_AT_encoding`, and the value should be `DW_ATE_signed`.
 - d. Call `dwarf_add_AT_unsigned_const` one more time to add the byte size. The attribute should be `DW_AT_byte_size`, and the value should be `4`.
2. Call `dwarf_new_die`. The tag is `DW_TAG_variable`, the parent is the CU DIE, the child is `NULL`, and the left and right siblings are `NULL`.
3. Call `dwarf_add_AT_name` to add the name `myvar` to the variable DIE.
4. Call `dwarf_add_AT_reference` to add the type DIE as an attribute of the variable DIE. The attribute is `DW_AT_type`.
5. Call `dwarf_add_AT_flag` to add the external flag to the variable DIE. The attribute is `DW_AT_external`, and the value is `true`.
6. Call the following operations:
 - `dwarf_new_expr` to create a location expression
 - `dwarf_add_expr_gen`, `dwarf_add_expr_addr`, `dwarf_add_expr_reg` to add operations to the location expression
 - `dwarf_add_expr_breg` to add operations to the location expression (optional)
 - `dwarf_add_AT_location_expr` to add the location expression to the variable DIE (the attribute is `DW_AT_location`)

All of the information about the global variable has been added to DIEs. Conversion is now complete.

Chapter 5. Using consumer and producer functions

This information shows how to create an application that both creates and uses DWARF debugging information. In most cases, DWARF debugging information will be produced by the z/OS XL C/C++ compiler. Therefore, most program analysis applications will need only the CDA consumer functions. However, if only ISD information is available, then the applications might need to use CDA producer functions to generate DWARF debugging information. For this reason, the sample code demonstrates the use of both CDA producer and consumer functions.

The example in this chapter uses the `libelf`, `libdwarf`, and `libddpi` libraries. It converts ISD debugging information to the DWARF format during run time by directly calling the converter function in `libddpi`. The example also shows how to use the `libdwarf` producer functions, once the DWARF debugging information becomes available. This example is not meant to be comprehensive.

Note: For more information about conversion, see Chapter 4, “Using producer APIs,” on page 39 and Chapter 6, “In Storage Debug (ISD) Information Conversion Utility,” on page 47.

The example files are delivered in the demo package, which is found in the `/usr/lpp/cbclib/source` directory. The package contains:

- `hello_isd.c`, a C-source file which will be compiled with the TEST compiler option
- `hello_dwarf.c`, a C source file which will be compiled with the DEBUG compiler option
- `demoa.s`, an assembler source, which implements a function to determine the size of a module loaded in storage
- `democ.c`, a C program, which demonstrates the use of functions of the CDA libraries
- `Makefile`, a makefile
- `README`, which is the basis of the content of this chapter

`hello_isd.c` and `hello_dwarf.c` create the program whose debugging information is the subject of this example. The two objects produced from these source files are linked into an HFS module (`hello`) which resides in the current directory.

`democ.c` contains the logic that demonstrates the use of the producer and consumer functions. `democ.c` will

- Load the `hello` module into storage.
- Create `libdwarf` consumer objects for all available debugging information.
- Print out the names of all global symbols found in the `hello` module.

Creating a consumer application with ISD conversion functionality

| If DWARF debugging information is available then the ELF object file can be used.
| If ISD information is available, a program analysis application can convert it into
| DWARF information by using the CDA ISD converter operations. If the debugging
| information is in neither format, then you must supply your own converter function.
| This information describes how to create a consumer application with conversion
| functionality.

Example: Process to create a consumer application with ISD converters

For more information about the CDA ISD converter operations, see "Conversion APIs" in *Common Debug Architecture Library Reference*.

The process for creating a consumer application with ISD conversion functionality is divided into three topics:

- "Initializing the libddpi environment"
- "Creating and using DWARF consumer objects" on page 45
- "Terminating the DWARF and ELF objects" on page 46

Note: The concepts and terms used in those topics are based on explanations in "Accessing debugging information" on page 18.

Initializing the libddpi environment

This information explains how to create and load a module, and set up the environment in order to use the libddpi operations.

Perform the following steps to create an application that converts ISD information into an ELF descriptor, then uses that descriptor.

1. Makefile compiles the `hello_isd.c` source file into the `hello_isd.o` object file, which contains ISD information. The object file resides in the current directory. For more information about the required compiler options, see "CDA requirements and recommendations" on page 7.
2. Makefile compiles `hello_dwarf.c` into the `hello_dwarf.o` object file and the `hello_dwarf.dbg` ELF object file. Only `hello_dwarf.dbg` contains the DWARF debugging information. Both files reside in the current directory. For more information about the required compiler options, see "CDA requirements and recommendations" on page 7.
3. Makefile links `hello_isd.o` and `hello_dwarf.o` into an HFS module (`hello`). Makefile now runs `democ.c` which controls the rest of this process.
4. The `hello` module is loaded into storage using the BPX1LOD USS Kernel interface.
5. The `__lmsize` assembler function determines the size of the `hello` module loaded in storage. This value will be used to create a `Ddpi_Space` object in step 8. `__lmsize` is implemented in the `demoa.s` assembler file.
6. operations are called to verify that the current versions of the DLLs meet or exceed the minimum required version. These operations are:
 - `elf_build_version`
 - `dwarf_build_version`
 - `ddpi_build_version`
7. `ddpi_init` initializes the libddpi environment. Before libddpi operations can be used, the environment must be initialized with `ddpi_init`. This creates a `Ddpi_Info` object, which holds information about the module loaded in storage.
8. `ddpi_space_create` creates a `Ddpi_Space` object which holds information about the `hello` module.
9. `ddpi_storagelocn_create` creates a storage location object (`Ddpi_StorageLocn`) which holds the storage-location information of the `hello` module.
10. `ddpi_storagelocn_get_space` obtains an associated space object from a given location object. The information about the module is kept in the space object,

so the space object is set as the module owner. In this example, the space object has just been created, and could immediately be set as the owner. However, it is more likely that ownership will be set after several objects have been created. The `Ddpi_StorageLocn` is the recommended interface to the `Ddpi_Space` object.

11. `ddpi_module_create` creates a `Ddpi_Module` object that represents the `hello` module.
12. `ddpi_class_create` creates a class object of type `Ddpi_CT_Program_code`. This class maps the portion of memory occupied by `hello`. Certain portions of memory occupied by the module are mapped according to their use, such as program code, WSA, or heap. `ddpi_class_create` is called to create a class object that maps the storage occupied by the program code, as this is the location of the debugging information.
13. `ddpi_entrypt_create` describes the entry point of the module. The entry point of the module is the key to finding the debugging information in the program code.
14. `ddpi_module_extract_C_CPP_information` goes through the module and identifies the CUs. This operation creates a list of `Ddpi_Elf` objects, each representing a CU found in the module. This includes CUs that have non-DWARF debugging information.

The consumer application can now start to create consumer objects.

Creating and using DWARF consumer objects

The `ddpi_module_extract_C_CPP_information` operation identifies each CU in the module. It is necessary to determine the format of the available debugging information. If DWARF debugging information is available, the ELF object file can be used. If ISD information is available, then it can be converted to DWARF using the ISD conversion operations. If the debugging information is in neither format, then you must supply your own conversion functions.

The following steps describe how to find CUs and create a `Dwarf_Debug` object for each of them.

1. `ddpi_elf_get_elf_file_name` queries the name of an ELF object file. If the executable module was compiled with the `DEBUG(FORMAT(DWARF))` compiler option, then an ELF object file has been created, and its name and location are stored in the CU. `ddpi_elf_get_elf_file_name` will retrieve this information. In this case, proceed to step 5.

If no file exists, the function returns `DW_DLV_NO_ENTRY`. For this example, this means that the information is in the ISD format. In general, this may not be the case, and additional logic is required to determine the kind of debugging information that is available. For more information on the possible types of debugging data, see “Accessing debugging information” on page 18.
2. `ddpi_elf_get_csect_addrs` retrieves the boundaries of the CU from the current ELF descriptor.
3. `ddpi_fp_convert_c_cpp_isdobj` converts the ISD debugging information. The ISD information is converted to the DWARF format using the CU boundaries.
4. `ddpi_elf_set_source` sets the source of the ELF descriptor associated with `hello`. The converted debugging information is kept in a temporary memory file. This can be seen as a temporary ELF object file, which will be used as the source of the ELF descriptor for the consumer process.

At this point, skip step 5 and proceed to step 6.

5. The name returned by `ddpi_elf_get_elf_file_name` is used to open the file, read the ELF information, and create an ELF descriptor. All character strings accepted and returned by the CDA libraries are in ASCII(ISO8859-1). The file name has to be converted to EBCDIC before calling `fopen`.
6. `dwarf_elf_init_b` initializes a `libdwarf` consumer object. Once all the CUs have been processed, a `libdwarf` consumer object (`Dwarf_Debug`) is initialized by calling `dwarf_elf_init_b`.
7. `ddpi_dealloc` frees the list of `Ddpi_Elf` objects. The list created by `ddpi_module_extract_C_CPP_information` is no longer needed.
8. `display_global_symbols` (a `democ.c` function) retrieves and prints out the global symbols found in `hello`. The debugging information is ready for consumption. This operation demonstrates a small subset of `libdwarf` operations that return the information to print out. More examples of DWARF operations can be found in the `dwarfdump` utility.

Terminating the DWARF and ELF objects

The main object of the example is now complete. The final steps show how to terminate the created objects.

1. `dwarf_get_elf` returns the ELF descriptors associated with the `libdwarf` consumer object.
2. `dwarf_finish` terminates the `libdwarf` consumer object. This function does not free all the storage used for ELF objects, which is why `dwarf_get_elf` was called before terminating the object.
3. `elf_end` terminates the ELF descriptor.
4. `ddpi_finish` releases any storage that was acquired while processing the module.

Chapter 6. In Storage Debug (ISD) Information Conversion Utility

In Storage Debug (ISD) information is produced by C/C++ compilers and other language translators to enable debugging tools to present information and aid developers in debugging. ISD information is not a programmable interface as the knowledge and understanding of the information is encapsulated in the debugging tools. This effectively limits the field of debug related tools. To remove this limitation a new form of debugging information has been introduced. The data uses the DWARF format, and is stored in ELF object files. For the convenience of the zSeries user, the debugging information can be accessed using the Common Debug Architecture (CDA) libraries and utilities. One of these utilities is the `isdcnvt` utility.

Prior to z/OS V1R6, the only method for generating debugging information was to use the TEST option to generate ISD information. As of z/OS V1R6, the DWARF debugging information is generated by using the DEBUG compiler option. However, DWARF debugging information can also be generated from ISD information by using `isdcnvt`.

The input to `isdcnvt` is an object file generated by the C/C++ compiler using the TEST or DEBUG(FORMAT(ISD)) compiler options. The utility produces a file containing the new debugging information which is suitable for use with debug tools that support ELF and DWARF interfaces, such as `dbx`.

The following syntax is used to invoke `isdcnvt`:

```
isdcnvt [-v] -o object_file_name
```

where:

- `-v` is an optional command line flag that produces version information for the `libelf`, `libdwarf`, and `libddpi` libraries
- *object_file_name* is the name of an object file that contains the ISD information

Object file formats supported by `isdcnvt` are OBJ, XOBJ and GOFF. Object files can have XPLINK or non-XPLINK linkage, but only object files produced by the IBM XL C/C++ compilers are currently supported.

Note: For more information about the supported compilers, see “CDA requirements and recommendations” on page 7.

The output file name is based on *object_file_name*. Although the object file name can have any suffix, only the standard `.o` suffix is recognized and replaced with the standard `.dbg` suffix when constructing the output file name. All other suffixes, including no suffix at all, are kept, and the standard `.dbg` suffix is appended when constructing the output file name.

Note: This process will overwrite any existing file with the same name as the expected output file.

`isdcnvt` is a UNIX System Services utility that runs in the shell environment. It supports only zFS files for input and output. If no errors are encountered during the conversion, the utility terminates with return code zero. If an error condition is detected during the conversion, the utility returns an error code with the following format:

CRR

where

- C is a decimal digit indicating the error code
- RR is a two-digit decimal number indicating the reason code

The error codes are:

- 1 - a recoverable error condition
- 2 - an internal error that should be reported to the IBM service team.

The reason codes associated with the error code 1 are:

- 01 - empty compilation unit
This error indicates that the compilation unit contained no code sections, which is typical for data-only compilation units. If this is an expected condition, the build process can check for this return code and continue processing.
- 02 - invalid usage
This error indicates that the utility was not invoked using the correct invocation syntax. To resolve the problem, ensure that the correct invocation syntax is used.

Note: The isdcnvt utility uses the `getopt()` run-time library function, which may emit error messages.
- 03 - failed to load debug APIs
To perform the conversion, the conversion utility requires debug APIs that are loaded at initialization. The APIs are provided in the CDAEED DLL, which is found in the CEE.SCEERUN2 MVS dataset. To resolve the problem, ensure that CDAEED is found by the loader using the MVS search order. For example, ensure that CEE.SCEERUN2 is in the STEPLIB environment variable.
- 04 - compilation unit has no debugging information
This error indicates that the compilation unit did not contain any debugging information. To resolve this problem, ensure that the compilation unit is compiled with the TEST or DEBUG(FORMAT(ISD)) compiler option.
- 05 - failed to open input file
This error can occur if an invalid object file has been specified, or if it does not have sufficient read permission. To resolve the problem, ensure that a valid object file is specified and that it has sufficient read permission.
- 06 - failed to open output file
An output file for the converted debugging information could not be opened. This can be caused by conditions such as insufficient space in the file system that is hosting the current directory, or no write permission for the current directory. To resolve the problem, ensure that the file system has sufficient space (usually one third of the input file size), and that the write permission is set for the current directory.
- 07 - version mismatch
The conversion utility dynamically loads debug APIs, so the version of the utility may not match the version of the debug APIs. To resolve the problem, ensure that the correct version of the debug APIs is found by the loader using the MVS search order.

The reason code associated with the error code 2 is a two-digit decimal number providing further information that can help diagnose the problem. This error code usually indicates a problem in the conversion utility or a language translator that produced the object file. To resolve this problem, contact IBM support and provide

the test case that reproduces the problem.

Chapter 7. Using the module map to improve performance

Given any program that is compiled with the DEBUG compiler option, the `dbgld` command can create a module map for the program. The module map associates each of the compiled program's functions, global variables, external types, and source files to the `.dbg` file that contains its debugging information.

A debugger that is written to use the module map will perform more efficiently for the following reasons:

- The start up time will be shorter, because the `.dbg` files are opened by CDA instead of the debugger. Only one `.dbg` file is loaded into memory at any given time, depending on which one is needed. The debugger also requires less memory because the entire DWARF debug instance is never loaded into memory at one time.
- Each `libdwarf` operation can complete an operation more quickly because the debugger needs to search only that information associated with the program element being debugged, instead of the entire DWARF debug instance.

Notes:

1. Debuggers that are written to use earlier versions of CDA will continue to load all of the debug side files (merged together into one large DWARF debug instance) at startup. There will be no significant change in startup time or operation execution time.
2. A debugger that can use the module map does not need to open the `.dbg` files or call the `elf_init_b()` or `dwarf_elf_init_b()` operation. CDA will do this automatically whenever it is required.
3. If using the module map, a debugger can set the DWARF error handler and error argument by calling `ddpi_info_set_dwarf_error_handler()`. This operation needs to be called only once, before the first call to any operation that returns a `Ddpi_Access` object.

Existing debuggers require considerable modification before they can make use of a module map. See "APIs that support use of the module map."

APIs that support use of the module map

The purpose of a `Ddpi_Access` object is to provide a way of accessing the DWARF debug information from DDPI. Because the debug information for each of the `Ddpi_Elf` objects can be accessed separately, each `Ddpi_Elf` object will be owned by a separate `Ddpi_Access` object. The `Ddpi_Module` object will contain the list of `Ddpi_Access` objects.

Table 5. Debugger tasks and the operations that execute them

Debugger task	Process
Look for the <code>Ddpi_Access</code> object that corresponds to a specific external type name.	<ol style="list-style-type: none">1. Call the <code>ddpi_module_list_type()</code> operation, passing in the type name, to get a list of type names in the module that match the given type name.2. Call the <code>ddpi_type_get_access()</code> operation, to retrieve the <code>Ddpi_Access</code> object for a specific external type name.

Table 5. Debugger tasks and the operations that execute them (continued)

Debugger task	Process
Look for the Ddpi_Access object that corresponds to a specific function.	<ol style="list-style-type: none"> 1. Call the <code>ddpi_module_list_function()</code> operation, passing in the function name, to get a list of functions in the module that match the given function name. 2. Call the <code>ddpi_function_get_access()</code> operation, to retrieve the <code>Ddpi_Access</code> object for a specific external function name.
Look for base type information for specific types that aren't in the current compilation unit.	Call the <code>ddpi_module_list_type()</code> operation, passing in the base type name.
Look for the Ddpi_Access object that corresponds to a specific source file.	<ol style="list-style-type: none"> 1. Call the <code>ddpi_module_list_sourcefiles()</code> operation, passing in the file name, to get a list of source files in the module that match the given file name. 2. Call the <code>ddpi_sourcefile_get_access()</code> operation to retrieve the <code>Ddpi_Access</code> object for the specific source file name.
Look for the Ddpi_Access object that corresponds to a specific global variable.	<ol style="list-style-type: none"> 1. Call the <code>ddpi_module_list_variable()</code> operation, passing in the variable name, to get a list of global variables in the module that match the given global variable name. 2. Call the <code>ddpi_variable_get_access()</code> operation to retrieve the <code>Ddpi_Access</code> object that corresponds to a specific global variable.
Look for the Ddpi_Access object that corresponds to a specific address in the loaded module (for example, when stopping at a breakpoint).	<ol style="list-style-type: none"> 1. Call the <code>ddpi_module_find_elf_given_address()</code> operation to specify the address of a breakpoint or other event. 2. Call the <code>ddpi_elf_get_owner()</code> operation to retrieve the <code>Ddpi_Access</code> object that is active at the step identified by the given address.
<p>Indicate which directories to search for <code>.dbg</code> or <code>.mdbg</code> files.</p> <p>Notes:</p> <ol style="list-style-type: none"> 1. This is necessary only if the <code>.dbg</code> or <code>.mdbg</code> files have been moved from their original location. 2. The <code>.mdbg</code> files are opened by <code>ddpi_module_extract_debug_info()</code>. 3. The <code>.dbg</code> files are opened by <code>ddpi_access_get_debug()</code> and <code>ddpi_access_get_dwarf_error()</code>, but only if their contents are not already in the <code>.mdbg</code> file. 	Call the <code>ddpi_info_set_dbg_dirs()</code> operation before any of the <code>.dbg</code> or <code>.mdbg</code> files need to be opened.

Table 5. Debugger tasks and the operations that execute them (continued)

Debugger task	Process
Retrieve the Dwarf_Error object from a Ddpi_Access object. Note: The debugger will need to pass a separate Dwarf_Error object to the libdwarf operations for each Dwarf_Debug instance.	Call the ddpi_access_get_dwarf_error() operation.
Set a DWARF error handler and error argument.	Call the ddpi_info_set_dwarf_error_handler() before the first call to any operation that returns a Ddpi_Access object.

Sample statements that illustrate use of a module map

This topic provides some sample statements that a debugger can use to extract debugging information for the function fun from the automatically generated module map.

Before using the code in Figure 8 on page 55, create the source files that it debugs, shown in Figure 5 and Figure 6.

```
/* hello.c */
int main() {
    return fun();
}
```

Figure 5. hello.c - The main module.

```
/* hello2.c */
int fun() {
    int a=5;

    return a;
}
```

Figure 6. hello2.c - Declaration of the function named "fun".

```

| 1 Ddpi_Module      module;
| 2 Ddpi_Error*     error;
| 3 Dwarf_Bool      mod_map;
| 4 Ddpi_Function*  function_list;
| 5 Dwarf_Unsigned  function_count;
| 6 Ddpi_Access     access;
| 7 Dwarf_Debug     dbg;
| 8 Dwarf_Error*    dwarf_error;

```

Notes:

1. Each `Ddpi_Module` object contains the list of `Ddpi_Access` objects for the compilation unit (main module). The `Ddpi_Module` object should be created prior to execution of the code in Figure 8 on page 55.
2. The `Ddpi_Error` object is a required parameter that handles error information generated by the producer or consumer application.
3. The `Dwarf_Bool` object indicates whether or not a module map was found for the main module.
4. Each `Ddpi_Function` object contains information about a specific function, including static functions. This object can be queried to get:
 - The fully qualified name of the function.
 - The unqualified name of the function.
 - The `Ddpi_Access` object that identifies the `.dbg` file for the function.
5. The `Dwarf_Unsigned` object contains the number of functions with a given name that are found.
6. The `Ddpi_Access` object provides a way of accessing the debugging information for the compilation unit in which the function is defined.
7. The `Dwarf_Debug` object contains the DWARF debugging information for the compilation unit in which the function is defined.
8. The `Dwarf_Error` object contains error information generated by DWARF operations.

Figure 7. Variables used in Figure 8 on page 55.

```

| 1 /* Call the extraction function for the module */
| ddpi_module_extract_debug_info(module, 0, &mod_map, error);
|
| 2 /* Locate the debugging information for the function named "fun" */
| ddpi_module_list_function(module, "fun", &function_list, &function_count, error);
|                                     2a                               2b
|
| 3 /* Get the Ddpi_Access object for "fun" */
| ddpi_function_get_access(function_list[0], &access, error);
|
| 4 /* Get the Dwarf debug instance from the Ddpi_Access object*/
| ddpi_access_get_debug(access, &dbg, error);
|
| 5 /* Get the Dwarf_Error object from the Ddpi_Access object */
| ddpi_access_get_dwarf_error(access, &dwarf_error, error);
|
| /* Since "fun" is a function in hello2.c, debugging can now
|    be done on any symbols in hello2.c.
| */

```

Notes:

1. Extract the debugging information from module. If the module map is found, mod_map is set.
2. Locate the debugging information for the function named "fun".
 - a. function_list should contain a single Ddpi_Function object because there is one function named "fun".
 - b. function_count should be equal to "1" because there is one function named "fun".
3. Get the Ddpi_Access object access from the first entry in function_list.
4. Get the Dwarf debug instance dbg from access.
5. Get the Dwarf_Error object error from access.

Figure 8. Statements for extraction of specific debugging information for a specific function

Appendix A. Diagnosing problems

This information tells you how to diagnose failures in the Common Debug Architecture (CDA) libraries and utilities. If you discover that the problem is a valid CDA problem, please refer to <http://techsupport.services.ibm.com/guides/handbook.html> for information on obtaining IBM service and support.

Using the 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 to confirm if the suspected failure is caused by an error in the CDA libraries and utilities, or by incorrect usage of them.

Step through each of the items in the diagnosis checklist below to see if they apply to your problem:

- Verify that your installation is at the most current maintenance level. That is, verify that you have received all issued IBM Program Temporary Fixes (PTFs) and have installed them. Your installation may have already received a PTF that fixes the problem.
- Check if the preventive service planning (PSP) bucket contains information related to your problem. The PSP is an online database available through IBM service channels. It gives information about product installation problems and other problems.
- Verify that the appropriate header files have been included and that the include paths are specified correctly, if the error occurs during compilation. That is:
 - Include `libelf.h` if `libelf` operations are called.
 - Include `libdwarf.h` and `dwarf.h` if `libdwarf` operations are called.
 - Include `libddpi.h` if `libddpi` operations are called.
- Verify that your application is compiled with the XPLINK compiler option if it calls `libddpi` operations. If your application is not compiled with the XPLINK compiler option, you will need to specify the runtime option `XPLINK(ON)` when executing your application.
- Verify that the sidedeck is included during the link step when linking your application. The `libelf` and `libdwarf` libraries are packaged for 31-bit as a single DLL module named `CDAEED` and for 64-bit as a single DLL module named `CDAEQED`. The `libddpi` library is packaged for 31-bit as a DLL module named `CDAEDPI` and for 64-bit as a DLL module named `CDAEQDPI`.
- Verify that `CDAEED` exists during the execution of your application. You can use the following code:

Note: `CDAEED` in the code sample below is a 32-bit library. If your application is 64-bit, replace `CDAEED` with `CDAEQED`.

```
#include <dll.h>
dllhandle*dllhand;
dllhand = dllload("CDAEED");
/*CDAEED is the name of the libdwarf/libelf DLL module */
if (dllhand ==NULL){
/*libdwarf/libelf DLL not found!*/
/*make sure CDAEED can be found
either through the STEPLIB or the LIBPATH */
}
```

- Verify that you are using the correct version of CDAEED. (If your application uses a libdwarf or a libelf header file that is incompatible with the CDAEED, your application might fail.) You can use the following code:

```

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 */
}

```

- If an abend occurs, then verify that it is caused by product failures and not by program errors. Read the CEEDUMP to determine if the abend happens within the CDA libraries or utilities. For example, the CEEDUMP would show if the exception occurred in the CDAEED load module for 31-bit or in the CDAEQED load module for 64-bit. Similarly, if the error occurred at an API entry point, then where the exception occurred would contain one or more of the keywords dwarf, elf, ddpi, dwarfdump, or isdcnvt.
- Consider writing a small test case that recreates the problem, after you identify the failure. The test case could help you determine if the error is in a user function or in CDA. 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 CDA failure, refer to the diagnosis procedures for the product that failed.
- If you are experiencing a no-response problem, try to force a dump, and cancel the program with the dump option.
- 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 CDA libraries. The following table lists how to find the level.

Library	API
libelf	elf_build_level
libdwarf	dwarf_build_level
libddpi	ddpi_build_level

Avoiding installation problems

Perform the following steps to avoid or solve most installation problems:

1. Review the step-by-step installation procedure for the Run-Time Library Extensions element. This documentation is located in the z/OS Program Directory.
2. Consult the PSP bucket as described in “Using the diagnosis checklist” on page 57.

If you still cannot solve the problem, develop a keyword string and contact your IBM Support Center.

You may need to reinstall CDA by using the procedure that is documented in the z/OS Program Directory. This procedure is tested for each product release and successfully installs the product.

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/](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.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director
IBM Canada Ltd. Laboratory
B3/KB7/8200/MKM
8200 Warden Avenue
Markham, Ontario L6G 1C7
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

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

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM	IBM logo	ibm.com®
AD/Cycle	BookManager	BookMaster
C/370	eServer	Language Environment
MVS	MVS/ESA	REXX
z/Architecture®	z/OS	zSeries

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java™ and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Linux® is a trademark of Linus Torvalds in the United States, other countries, or both.

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

Standards

The `libddpi` library supports the DWARF Version 3 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.

Bibliography

This bibliography lists the publications for IBM products that are related to Common Debug Architecture. It includes publications covering the application programming task.

The bibliography is not a comprehensive list of the publications for these products, however, it should be adequate for most z/OS CDA users. Refer to *z/OS Information Roadmap, SA22-7500*, for a complete list of publications belonging to the z/OS product.

Related publications not listed in this section can be found in *z/OS DFSMS™ Using the New Functions, SC26-7473*, in *z/OS Collection, SK3T-4269*, or on a tape that is available with z/OS.

z/OS Run-Time Library Extensions

- *DWARF/ELF Extensions Library Reference, SC09-7655*
- *z/OS Common Debug Architecture Library Reference, SC09-7654*
- *C/C++ Legacy Class Libraries Reference, SC09-7652*

z/OS

- *z/OS Introduction and Release Guide, GA22-7502*
- *z/OS Planning for Installation, GA22-7504*
- *z/OS Summary of Message and Interface Changes, SA22-7505*
- *z/OS Information Roadmap, SA22-7500*

z/OS XL C/C++

- *z/OS XL C/C++ Programming Guide, SC09-4765*
- *z/OS XL C/C++ User's Guide, SC09-4767*
- *z/OS XL C/C++ Language Reference, SC09-4815*
- *z/OS XL C/C++ Messages, SC09-4819*
- *z/OS XL C/C++ Run-Time Library Reference, SA22-7821*
- *z/OS XL C Curses, SA22-7820*
- *z/OS XL C/C++ Compiler and Run-Time Migration Guide for the Application Programmer, GC09-4913*
- *Standard C++ Library Reference, SC09-4949*
- *IBM Open Class® Library Transition Guide, SC09-4948*

z/OS Language Environment

- *z/OS Language Environment Concepts Guide, SA22-756*
- *z/OS Language Environment Customization, SA22-7564*
- *z/OS Language Environment Debugging Guide, GA22-7560*
- *z/OS Language Environment Programming Guide, SA22-7561*
- *z/OS Language Environment Programming Reference, SA22-7562*
- *z/OS Language Environment Run-Time Application Migration Guide, GA22-7565*

- *z/OS Language Environment Writing Interlanguage Communication Applications*, SA22-7563
 - *z/OS Language Environment Run-Time Messages*, SA22-7566
-

z/Architecture

- *z/Architecture Principles of Operations*, SA22-7832 which is available at:
www-03.ibm.com/servers/eserver/zseries/zos/bkserv/zswpdf/zarchpops.html

Index

A

- accessing DIEs 37
- addresses
 - relocation 39
- addresses in memory image 39
- API types, libddpi
 - CDA-application model 3
 - conversion 3
 - DWARF-expression 3
 - support 3
 - system-dependent 3
 - system-independent 3
- APIs
 - consumer 2
 - producer 2
- application module, extracting debugging information 18
- ASCII
 - codeset 7
 - compiler option 7

C

- CDA
 - definition 1
 - libraries 2
- changes 6
 - CDA 6
- checklist 57
- codeset
 - ASCII(ISO8859-1) 7
- Common Debug Architecture 1
- compiler options
 - ASCII 7
 - DEBUG 18, 51
 - GONUMBER 6
 - NOTEST 6
 - TEST 6
 - XPLINK 7
- compiler version requirements 7
- consumer
 - API 2
 - example 43
 - object 18
- consuming a DWARF object 36
- conversion
 - application 39, 43
 - direct function calls 18
 - supported formats 47
 - symbol 40
 - utility 6

D

- DEBUG compiler option 18, 51
- debugging information
 - converting 18

- debugging information (*continued*)
 - non-DWARF 18
 - read from ELF descriptor 14, 15
 - testing for DWARF format 18
 - write to ELF descriptor 12
- descriptor 9
- DIEs
 - accessing 37
 - navigating 37
 - traversing 37
- DWARF
 - consumer object 36
 - definition 1
 - format 3
 - objects 1
 - producer object 14
- Dwarf_Debug 1
- Dwarf_P_Debug 1
- dwarfdump 6

E

- ELF 2
 - definition 1
 - ELF
 - descriptor 9
 - object file, definition 1
 - object file, loading 39
 - object file, read from 15
 - read from descriptor 14
 - using a descriptor 15
 - write to descriptor 12
- ELF file
 - relocating addresses 23
- ELF file structure 23
- ELF files
 - relocating addresses 22
- ELF symbol structure 23
- ELF symbol table, loading 23
- elfload.c
 - relocating addresses within an ELF file 23
- error codes, isdcnvt 47
- examples, location viii
- Executable and Linking Format 2
- existing debuggers
 - modifying to use the module map 51

G

- global variable integer 41
- GONUMBER compiler option 6

H

- HEAPOOLS(on) run-time option 7

I

- In Store Debug 6
- initializing libelf and libdwarf 21
- ISD 6
- isdcnvt 6
 - error codes 47
 - options 47
 - supported object file formats 47
 - syntax 47

K

- keyboard 59

L

- libddpi library 3
- libdwarf library 3
- libdwarf objects definition 1
- libelf library 2
- libraries
 - CDA 2
 - interaction overview 9
 - libddpi 3
 - libdwarf 3
 - libelf 2
 - using libelf and libdwarf 12, 14, 21, 39
 - using libelf, libdwarf, and libddpi 15, 43
- location expression 41

M

- MD5 signature
 - and relocation of addresses within an ELF file 23
- module map
 - description 51

N

- navigating DIEs 37
- non-DWARF debugging information 18
- NOTEST compiler option 6

O

- object
 - consumer 1, 15
 - DWARF 1
 - ELF object file 1
 - libdwarf 1
 - producers 1
- options
 - compiler 6, 7
 - DEBUG 51
 - isdcnvt 47
 - run-time 7

P

- performance
 - enhancement, as of z/OS V1R10 51
- PPA1 section 39
- PPA2 section 39
- producer
 - API 2
 - example 39

R

- read
 - DWARF debugging information 14, 15
 - from ELF descriptor 14
 - from ELF object file 15
- relocation 39
 - of addresses within an ELF file 22
- relocation array 22
- reporting failures 57
- requirements
 - CDA 7
 - compiler 7
 - user v
- run-time option
 - HEAPPOOLS(on) 7

S

- sample applications
 - consumer 21, 43
 - dwarfdump 6
 - producer 39
- samples
 - elfload.c 23
- shortcut keys 59
- standards
 - DWARF 3
 - ELF 2
- supported object file formats 47
- symbol, conversion 40

T

- tasks
 - avoiding installation problems
 - steps for 58
 - converting a global variable integer
 - steps for 41
 - converting a symbol
 - steps for 40
 - creating a line-number table
 - steps for 39
 - preparing a .debug_ppa section
 - steps for 39
- technical support, finding ix
- terminating libelf and libdwarf 38
- TEST compiler option 6
- testing for DWARF debugging information 18
- traversing DIEs 37

U

- user area 7
- user interface
 - accessibility 59
 - disability 59
- user requirements v
- using DWARF object 36
- utilities
 - dwarfdump 6
 - isdcnvt 6

V

- variable-length user area 7

W

- write
 - DWARF debugging information 12
 - to ELF descriptor 12

X

- XPLINK compiler option 7



Program Number: 5694-A01

Printed in USA

SC09-7653-02

