z/OS

Common Debug Architecture User's Guide

SC09-7653-01

z/OS

Common Debug Architecture User's Guide

SC09-7653-01

Note!
Before using this information and the product it supports, be sure to read the information in “Notices” on page 37.

Second Edition (September 2004)

This edition applies to Version 1 Release 6 of z/OS C/C++ (5694-A01), Version 1 Release 6 of z/OS.e (5655-G52), and to all subsequent releases until otherwise indicated in new editions.

Ensure that you apply all necessary PTFs for the program.

Order publications through your IBM representative or the IBM branch office serving your location. Publications are not stocked at the address below. You can also browse the documents on the World Wide Web by clicking on ″The Library″ link on the z/OS home page.

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

Include the title and order number of this document, and the page number or topic related to your comment. When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

About this document

This document introduces the user to Common Debug Architecture (CDA). It 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.

Who should use this document

This document is intended for programmers who will be developing program analysis applications and debugging applications for the IBM® 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 V1R6 operating system.

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

- The z/OS operating system
- The `libdwarf` APIs
- The `libelf` APIs
- The ELF ABI
- Writing debugging programs in C on z/OS
- Writing debugging programs in C++ on z/OS
- POSIX on z/OS
- The Language Environment® (LE) on z/OS
- UNIX® System Services (USS) shell on z/OS

A note about examples

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

CDA and related publications

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

Table 1. CDA, DWARF, and ELF publications

Document title and number	Key sections/chapters in the document
z/OS Common Debug Architecture Library Reference, SC09-7654	The reference for IBM's `libddpi` library. It includes: • General discussion of CDA • APIs and data types related to stacks, processes, operating systems, machine state, storage, and formatting This book is available at: 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
DWARF/ELF Extensions Library Reference, SC09-7655	The reference for IBM's extensions to the `libdwarf` and `libelf` libraries. It includes: • Consumer APIs • Producer APIs This document discusses only these extensions, and does not provide a detailed explanation of DWARF and ELF. This book is available at: http://www.ibm.com/software/awdtools/libraryext/library/.
System V Application Binary Interface Standard	The Draft April 24, 2001 version of the ELF standard. For more information on this book, go to: http://www.ibm.com/software/awdtools/libraryext/library/.
ELF Application Binary Interface Supplement	The Draft April 24, 2001 version of the ELF standard supplement. For more information on this book, go to: http://www.ibm.com/software/awdtools/libraryext/library/.
DWARF Debugging Information Format, Version 3	The Draft 8 (November 19, 2001) version of the DWARF standard. This document is available on the web.
Consumer Library Interface to DWARF	The revision 1.48, March 31, 2002, version of the `libdwarf` consumer library. This book is available at: http://www.ibm.com/software/awdtools/libraryext/library/.
Producer Library Interface to DWARF	The revision 1.18, January 10, 2002, version of the `libdwarf` producer library. This book is available at: http://www.ibm.com/software/awdtools/libraryext/library/.
MIPS Extensions to DWARF Version 2.0	The revision 1.17, August 29, 2001, version of the MIPS extension to DWARF. This book is available at: http://www.ibm.com/software/awdtools/libraryext/library/.
z/OS C/C++ User's Guide, SC09-4767	Guidance information for: • z/OS C/C++ examples • Compiler options • Binder options and control statements • Specifying z/OS Language Environment run-time options • Compiling, IPA linking, binding, and running z/OS C/C++ programs • Utilities (Object Library, CXXFILT, DSECT Conversion, Code Set and Locale, ar and make, BPXBATCH) • Diagnosing problems • Cataloged procedures and REXX EXECs supplied by IBM This book is available at: http://www.ibm.com/software/awdtools/czos/library.
z/OS C/C++ Programming Guide, SC09-4765	Guidance information for: • Implementing programs that are written in C and C++ • Developing C and C++ programs to run under z/OS and z/OS.e • Using XPLINK assembler in C and C++ applications • Debugging I/O processes • Using advanced coding techniques, such as threads and exception handlers • Optimizing code • Internationalizing applications

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

Table 2. Publications by task

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

Softcopy documents

The z/OS™ Common Debug Architecture publications are supplied in PDF formats and 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 Acrobat Reader. If you do not have the Adobe Acrobat Reader, you can download it for free from the Adobe Web site at http://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 BookManager®, see *z/OS Information Roadmap*.

Softcopy examples

Most of the larger examples in the Common Debug Architecture documents are available in machine-readable form in the directory `/usr/lpp/cbclib/source`.

Common Debug Architecture on the World Wide Web

Additional information on Common Debug Architecture is available on the World Wide Web on the Common Debug Architecture home page at:
http://www.ibm.com/software/awdtools/commondebug

This page contains late-breaking information about Common Debug Architecture. There are links to other useful information, such as the Common Debug Architecture information library.

Where to find more information

Please see *z/OS Information Roadmap* for an overview of the documentation associated with z/OS, including the documentation available for z/OS Language Environment.

Accessing z/OS licensed documents on the Internet

z/OS licensed documentation is available on the Internet in PDF format at the IBM Resource Link™ Web site at:

`http://www.ibm.com/servers/resourcelink`

Licensed documents are available only to customers with a z/OS license. Access to these documents requires an IBM Resource Link user ID and password, and a key code. With your z/OS order you received a Memo to Licensees, (GI10-0671), that includes this key code. [1]

To obtain your IBM Resource Link user ID and password, log on to:

`http://www.ibm.com/servers/resourcelink`

To register for access to the z/OS licensed documents:

1. Sign in to Resource Link using your Resource Link user ID and password.
2. Select **User Profiles** located on the left-hand navigation bar.

Note: You cannot access the z/OS licensed documents unless you have registered for access to them and received an e-mail confirmation informing you that your request has been processed.

Printed licensed documents are not available from IBM.

You can use the PDF format on either **z/OS Licensed Product Library CD-ROM** or IBM Resource Link to print licensed documents.

Using LookAt to look up message explanations

LookAt is an online facility that lets you look up explanations for most messages you encounter, as well as for some system abends and codes. Using LookAt to find information is faster than a conventional search because in most cases LookAt goes directly to the message explanation.

You can access LookAt from the Internet at:
http://www.ibm.com/eserver/zseries/zos/bkserv/lookat/ or from anywhere in z/OS or z/OS.e where you can access a TSO/E command line (for example, TSO/E prompt, ISPF, z/OS UNIX System Services running OMVS).

The LookAt Web site also features a mobile edition of LookAt for devices such as Pocket PCs, Palm OS, or Linux-based handhelds. So, if you have a handheld device with wireless access and an Internet browser, you can now access LookAt message information from almost anywhere.

To use LookAt as a TSO/E command, you must have LookAt installed on your host system. You can obtain the LookAt code for TSO/E from a disk on your *z/OS Collection* (SK3T-4269) or from the LookAt Web site's **Download** link.

---

1. z/OS.e customers received a Memo to Licensees, (GI10-0684) that includes this key code.

Chapter 1. About Common Debug Architecture

This chapter gives an overview of each of the components of Common Debug Architecture (CDA) and the requirements and restrictions for an application developer who uses those components.

CDA components are based on the Executable and Linking Format (ELF) application binary interfaces (ABIs) and the DWARF format.

CDA was introduced in z/OS V1R5 to provide a consistent format for debug information on z/OS. The CDA libraries provide a set of APIs to access this debug information. These APIs will help support the development of debuggers and other program analysis applications for z/OS. CDA is intended to provide an opportunity to work towards a common debug information format across the various languages and operating systems that are supported on the zSeries™ eServer platform. The architecture is implemented in the z/OS CDA libraries component of the z/OS Run-Time Library Extensions element of z/OS V1R5 and higher.

The core of the Common Debug Architecture is the DWARF standard. DWARF is an industry-standard and language-independent format for debugging information. It is designed to meet the symbolic, source-level debugging needs of different languages in a unified fashion. The design of the debugging information format is open-ended, allowing for the addition of new debugging information to accommodate new languages or debugger capabilities. 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.

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 platforms that also use DWARF.

CDA-compliant applications store the DWARF debugging information in a separate ELF object file. Because this debug information is not in a typical object file, both of the following are minimized when the executable module is loaded into memory.

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

Using a separate object file enables the program analysis application to load specific information only if it is needed. If you create the separate ELF object file with the `DEBUG` option of the z/OS C/C++ compiler, the file has a `*.dbg` extension.

Note: Whenever this document refers to an ELF object file, it is discussing this separate ELF object file that stores only DWARF debugging information.

CDA libraries and utilities

CDA comprises three libraries and two utilities.

Libraries are:

- `libelf`
- `libdwarf`

- libddpi

Utilities are:

- isdcnvt
- dwarfdump

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

The libddpi library is available in XPLINK object form only.

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 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 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/cbclib/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/cbclib/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/cbclib/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/cbclib/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 zSeries eServer C/C++ languages
- Future support for z/OS and zSeries eServer languages such as FORTRAN, HLASM, COBOL, and PL/I

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`, refer to *z/OS Common Debug Architecture Library Reference*.

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

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/cbclib/include/libddpi/ directory.
- Compile with the XPLINK option and bind with libddpi.a.

  The libddpi library is packaged as the static library libddpi.a, which can be located in the /usr/lpp/cbclib/lib/ directory. The member libddpi.a contains both XPLINK 31–bit and 64–bit versions of the libddpi library.

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

API groups	Description
CDA application model APIs	This group allows developers to model applications they are analyzing and to keep track of debugging information with that model.
Support APIs	This group provides information about system settings in a universal manner.
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.
Conversion APIs	This group helps convert ISD debugging information into DWARF debugging information.

isdcnvt

Note: isdcnvt cannot be used to convert 64-bit objects. Debug information for 64–bit 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 C/C++ compilers. For more information, see “CDA limitations” on page 6.

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 C/C++ compiler `TEST` option.

For more information on `isdcnvt`, see Chapter 6, "In Storage Debug (ISD) Information Conversion Utility," on page 31. For more information on the conversion APIs, see *z/OS 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 V1R6

CDA libraries shipped with z/OS V1R6 include additional codeset-conversion APIs. These APIs handle the transfer of information between ELF objects that use strings encoded in the ISO8859-1 codeset, and applications on the z/OS platform that use string literals encoded in EBCDIC codeset. For more information, see the codeset-conversion API information in *z/OS Common Debug Architecture Library Reference* and the initialization and termination consumer API information in *DWARF/ELF Extensions Library Reference*.

The `LIBELF_DLL_VERSION` defined in `libelf.h` has been updated to `02002002` 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 that passes 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 probably indicates an attempt to use the code with an earlier version of ELF/DWARF support. Applications built with the current version of a DLL are not completely compatible with previous versions of that DLL.

Note: Future ELF/Dwarf DLLs will be backward compatible with the current versions. This means that any application built with a current version of a DLL will continue to work, as current DLLs are updated.

CDA requirements and recommendations

The CDA libraries are compiled with the z/OS 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 codeset 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 codeset. You can use the codeset conversion APIs to change the codeset. For more information, see *z/OS Common Debug Architecture Library Reference*, SC09-7654. For more information about the z/OS C/C++ compiler options, see *z/OS 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:
  - C/C++ for MVS/ESA V3R2 or greater
  - Any release of OS/390 C/C++
  - Any release of z/OS C/C++

  This support is not intended to work with debugging information generated by the C/370™ or 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 chapter discusses how the `libelf`, `libdwarf`, and `libddpi` libraries work together to access and use debugging information. It 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 chapter is divided up into the following sections:

Section	Description
"Creating an ELF descriptor"	This section explains how `libelf` uses a file handle and creates an ELF descriptor.
"Writing DWARF data to the ELF object file" on page 10	This section explains how `libelf` and `libdwarf` add debugging information from the ELF descriptor to the ELF object file.
"Reading from an ELF object file with libelf and libdwarf" on page 11	This section explains how `libelf` and `libdwarf` use the debugging information from the ELF object file.
"Reading from an ELF object file with libelf, libdwarf, and libddpi" on page 12	This section explains how `libelf`, `libdwarf`, and `libddpi` use the debugging information from the ELF object file.
"Accessing debugging information" on page 15	This section explains how to interrupt the consuming process in order to convert non-DWARF debugging information.

Creating an ELF descriptor

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

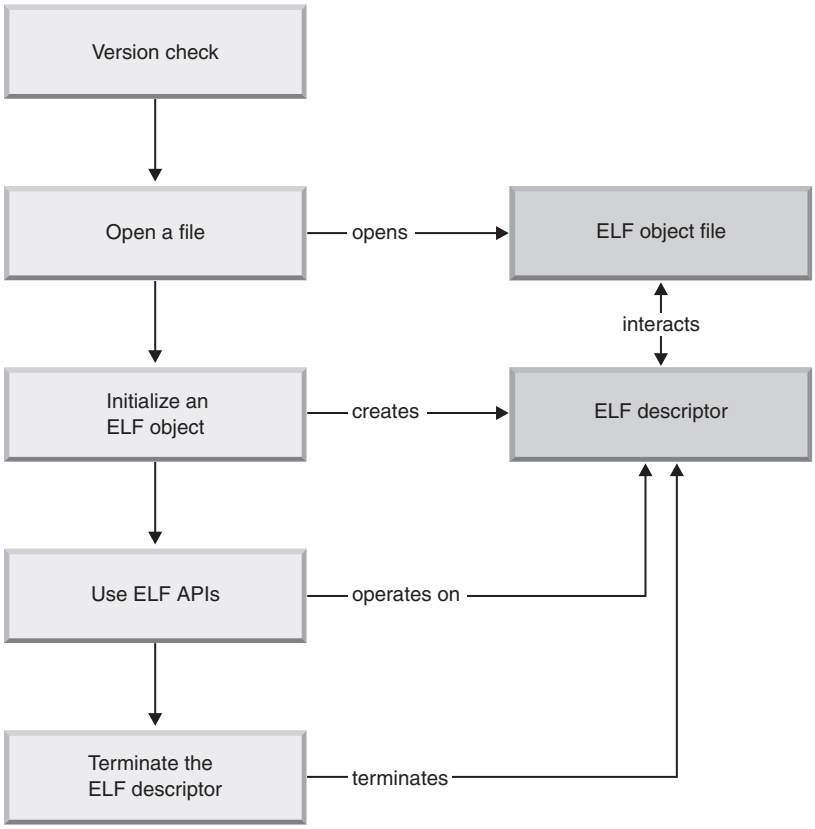

Figure 1. Creation of an ELF descriptor

The following stages show how producer or consumer functions create an ELF descriptor with calls to `libelf` functions.

Stage	Description

Version check	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:
	`#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`
	`    */`
	`  }`
	`}`
	
	It is mandatory to perform a verification of the ELF version before using the other functions offered by `libelf`. For example:
	`  /* 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 */`
	`}`

Open a file	The producer or consumer functions create a file handle for the ELF object file. This file handle is used to create an ELF descriptor. Consult *z/OS C/C++ Run-Time Library Reference* for more information on opening files and creating file handles.

Initialize ELF descriptor	An ELF descriptor is required before you can call any other `libelf` functions. The file handle is used to initialize `libelf` and create an ELF descriptor for the ELF object file. Which `libelf` function is used depends on which function is used to create a file handle. For example, if the file handle is created using `fopen`, then `elf_begin_b` is used. The following code demonstrates how to use the file pointer obtained from `fopen` to create the ELF descriptor:
	`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);`

Operate on the descriptor	After the ELF descriptor is initialized, you are free to call any of the `libelf` functions. For example, `elf_getscn` returns an ELF section, and `elf_kind` describes that section.
Terminate ELF descriptor	When finished with the debugging information, the descriptor is terminated with `elf_end`. **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.

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.

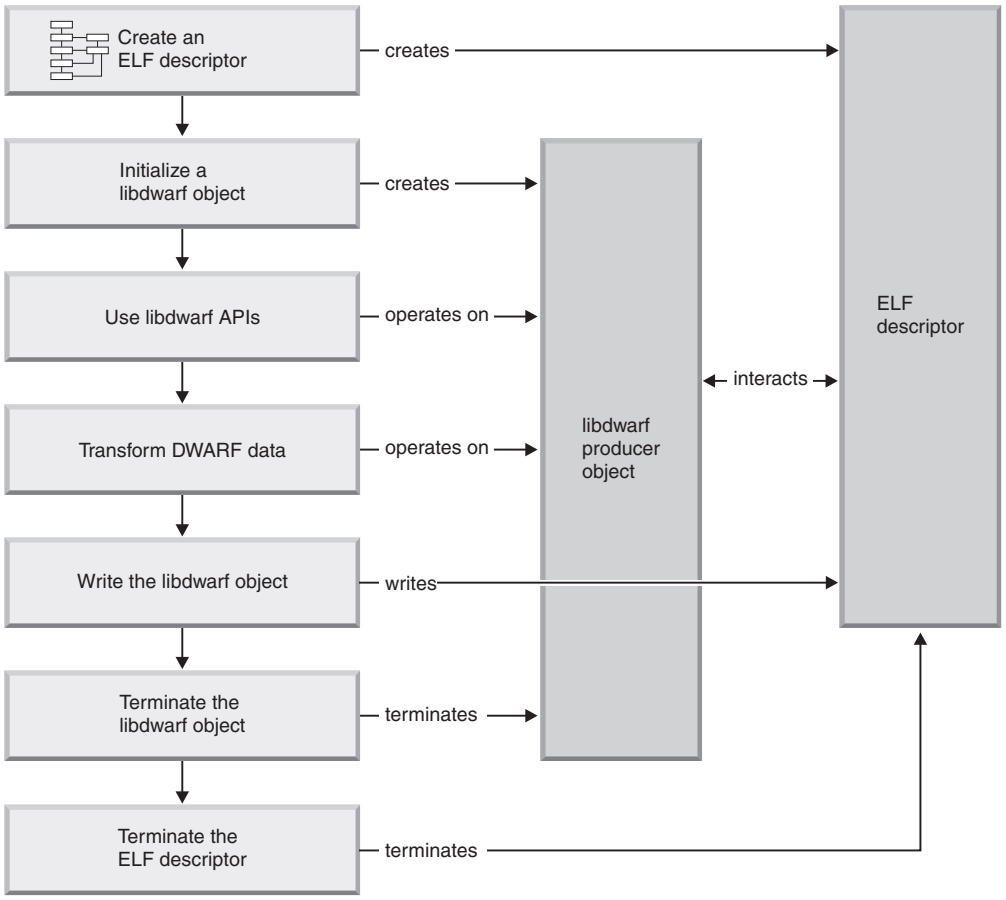

Figure 2. Write to an ELF object file

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

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 7.

Stage	Description

Initialize a `libdwarf` object	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.
	
	The following code demonstrates how to initialize the DWARF producer object:
	`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`
	`);`
	
	Note: The `ehdr` is extracted from the descriptor. An update to the header will update the descriptor.
	`/* 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);`

Use `libdwarf` APIs	`libdwarf` producer functions 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 functions can use it to read the DWARF debugging information from the ELF object file. This section discusses how consumer functions read from an ELF object file using the `libelf` and `libdwarf` libraries.

The following diagram shows an overview of the process.

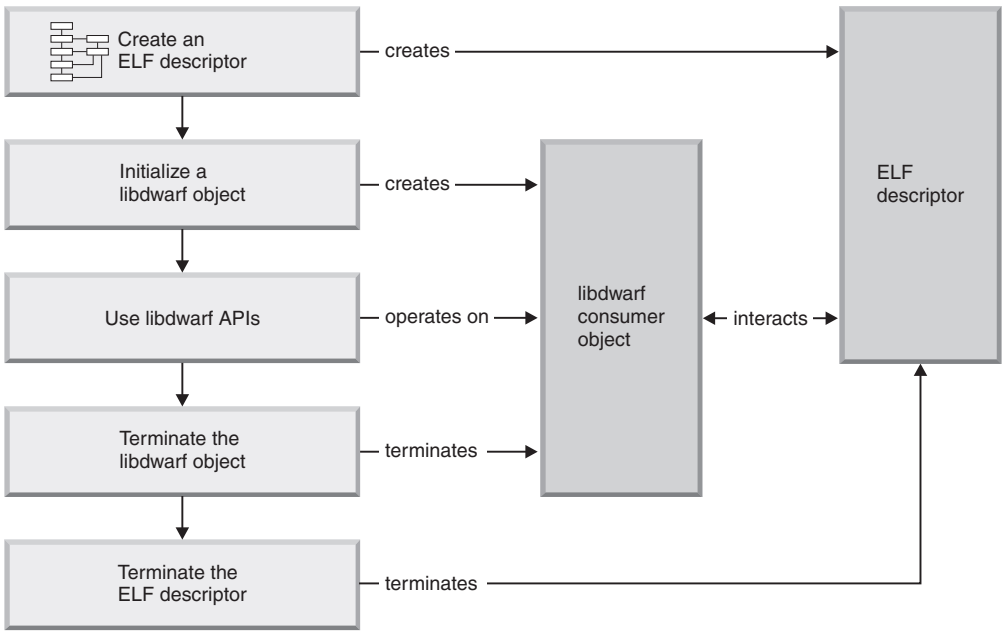

Figure 3. Read from an ELF object file with libelf and libdwarf

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

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 7.
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` functions 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 functions can use it to read the DWARF debugging information from the ELF object file. This section discusses how consumer functions 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 11.

The following diagram shows an overview of the process.

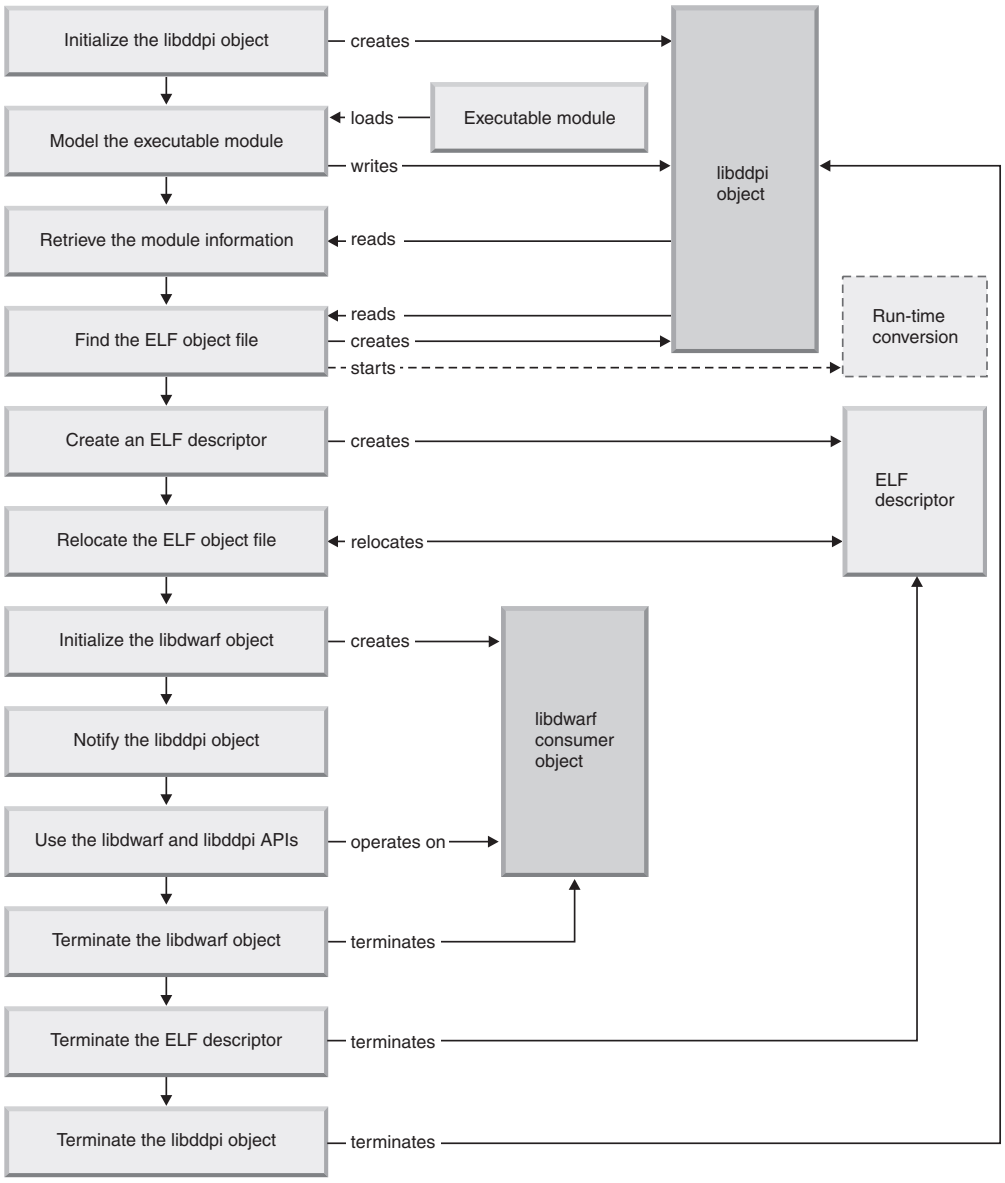

Figure 4. Read from an ELF object file with libelf, libdwarf, and libddpi

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

Stage	Description
Initialize `libddpi` object	Call `ddpi_init` to create a `Ddpi_Info` object. `Ddpi_Info` is a starting point that tracks: • The objects that model the application environment • The ELF object(s) • The DWARF object(s)

Stage	Description
Model the executable module	Use `libddpi` functions to retrieve information from the executable module. For example: • `ddpi_space_create` represents the address space in which the executable module resides • `ddpi_storagelocn_create` provides access to user storage • `ddpi_module_create` represents the actual executable module • `ddpi_entrypt_create` represents the entry point of the executable module
Retrieve module information	Call `ddpi_module_extract_C_CPP_information`. This function identifies all the compilation units (CU) in the executable module, then creates a `Ddpi_Elf` object to represent each CU. Each object holds the necessary information to load the DWARF debugging information that is in the CU.
Find ELF object file	Call `ddpi_elf_get_elf_file_name` to search for the name of an ELF object file. If the file name can not be found it returns `DW_DLV_NO ENTRY`, which indicates that this CU is not compiled with the `DEBUG(FORMAT(DWARF))` 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” on page 15.
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 7
Relocate ELF object file	Call `ddpi_elf_load_cu` 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 debug_ppa section” on page 21.
Initialize the `libdwarf` object	Initialize the `Dwarf_Debug` consumer object by calling `dwarf_elf_init` and using the ELF descriptor.
Notify `libddpi` object about `libdwarf` object.	Call `ddpi_access_set_debug` to let the `Ddpi_Info` object know about the newly created DWARF consumer object. This is done only once per module/program object.
Use `libdwarf` and `libddpi` functions	`libdwarf` functions are called to retrieve the DWARF data. For example, `dwarf_get_globals` will retrieve the list of global symbol entries, and `ddpi_module_get_major_name` will retrieve the major name from the given `Ddpi_Module` object.
Terminate the `libdwarf` object	`dwarf_finish` is called to terminate the `Dwarf_Debug` object.
Terminate ELF descriptor	The ELF descriptor is terminated with `elf_end`.
Terminate the `libddpi` objects	`ddpi_finish` is called to terminate the `Ddpi_Info` object.

Accessing debugging information

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

- Debugging information compiled with IBM z/OS C and C++ compilers
- Debugging information compiled with another compiler

The `libddpi` library contains functions that have been created specifically to set up access to executable modules created with the z/OS C/C++ compilers. Accordingly, most of this section 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 functions. For more information, see "Accessing non-z/OS C/C++ debugging information" on page 16.

Accessing z/OS C/C++ debugging information

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

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

- 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 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 C/C++ User's Guide*.

If a CU object was created with the DEBUG compiler option, the `ddpi_elf_get_elf_file_name` API 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" on page 16.

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

- 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 21.

Accessing and converting ISD information

CDA defines consumer functions 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 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 31.

- `libddpi` conversion APIs

  The `ddpi_convert_c_cpp_isdobj` and `ddpi_fp_convert_c_cpp_isdobj` functions 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 C/C++ debugging information

Extraction is started by calling the `ddpi_module_extract_C_CPP_information` function. If the executable module was not compiled with the IBM z/OS 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 `Ddpi_Elf` objects. For more information on how to create a converter application, see Chapter 4, "Using producer APIs," on page 21.

Chapter 3. Using consumer functions

This chapter 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.

Note: This chapter 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
- Terminating `libelf` and `libdwarf`

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

Initializing libelf and libdwarf

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

Initialization process

The following is an overview of the `libdwarf` initialization process.

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

The application uses the `elf_begin` function to create an ELF descriptor. This function 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 functions that can be used are `elf_begin_b` and `elf_begin_c`. Consult the `libelf` documentation for details on using these functions.

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 functions. 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_DLV_OK on successful completion. It returns DW_DLV_ERROR if an error occurs. dwarf_elf_init returns DW_DLV_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.

Consuming DWARF data

Once a Dwarf_Debug object has been created its data may be used by the application. This section discusses how the application uses libdwarf functions to extract information from its DWARF objects. That is, it discusses how to:

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

Traversing the DIE hierarchy

This section describes how to traverse the DIE hierarchy in the .debug_ppa section. This can be used as an example for traversing 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 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 using the `dwarf_child` function as follows:

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

The `in_die` variable is the root DIE. The application continues processing children until the above call returns `DW_DLV_NOENTRY` indicating that it has reached the bottom of the hierarchy.

The application now proceeds to traverse the siblings of the root DIE with the `dwarf_siblingof` function. For example:

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

Accessing information in a DIE

This section lists the `libdwarf` functions used by application to access data within a DIE.

Table 3. DIE access functions

Call	Description

`dwarf_tag(`
`  die, &tag, &err`
`);`
	This call retrieves the TAG of a DIE.

`dwarf_diename(`
`  dbg, &tagname, &err`
`);`
	This call retrieves the name of a TAG.

`dwarf_dieoffset(`
`  die, &overall_offset, &err`
`);`
	This call retrieves the overall offset of a DIE.

`dwarf_die_CU_offset(`
`  die, &offset, &err`
`);`
	This call retrieves the offset of a DIE within a given compilation unit.

`dwarf_attrlist(`
`  die, &atlist, &atcnt, &err`
`);`
	This call retrieves a list of the attributes for a DIE.

`dwarf_formudata(`
`  attrib, &val, &err`
`);`
	This call retrieves the unsigned value of a given attribute.

`dwarf_whatform(`
`  attrib, &theform, &err`
`);`
	This call retrieves the form of a given attribute.

Terminating libelf and libdwarf

This section discusses how the application terminates its interaction with `libdwarf` and the associated ELF descriptors. Termination is done in the opposite order from initialization. That is, first the `Dwarf_Debug` object is terminated, then the ELF descriptor.

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

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

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

```
elf_end(elf);
```

Chapter 4. Using producer APIs

This chapter 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 example is similar to the `isdcnvt` utility which uses the `libddpi` conversion functions.

Note: This chapter 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 sections:

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

Steps for converting 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
   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_lne_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_lne_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 section assumes that each CU has a corresponding ELF object file, and that the file contains the addresses of object information within the CU. When the CU is loaded into a memory image, the addresses of the object information are changed to available memory addresses. When the ELF object file is loaded, the addresses 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.

This section shows how to record the address of the PPA1 and PPA2 blocks in the CU, and how to create an MD5 signature for the ELF object.

Complete the following steps to create a `.debug_ppa` section.

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 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. Call `dwarf_new_die`. If a DIE is initially created with a NULL parent, it can be linked later by calling `dwarf_die_link`.

---

2. Add the applicable attributes to the DIE. Attributes are added by calling the following functions:
   - `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 23.
   - `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 `dwarf_new_expr`. Operations are added to the location expression by calling the following functions:

- `dwarf_add_expr_gen` is a generic function 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 of converting a symbol

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

This example shows how to convert a global variable integer (`myvar`).

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 a 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 functions:
   - `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 chapter 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 C/C++ compiler. Therefore, most applications will need only the CDA consumer functions. However, if only ISD information is available, then the applications may 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 21 and Chapter 6, “In Storage Debug (ISD) Information Conversion Utility,” on page 31.

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 conversion

This example is divided into three sections:

- “Initializing the libddpi environment” on page 26
- “Creating and using consumer objects” on page 27
- “Terminating the objects” on page 28

Note: The concepts and terms used in this section are based on explanations in “Accessing debugging information” on page 15.

Initializing the libddpi environment

This section explains how to create and load a module, and set up the environment in order to use the `libddpi` functions.

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

1. `Makefile` compiles `hello_isd.c` into the `hello_isd.o` object file, which contains ISD information. The file resides in the current directory. For more information about the required compiler options, see “CDA requirements and recommendations” on page 5.

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 5.

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. The following functions are called to verify that the current versions of the DLLs meet or exceed the minimum required version:
   - `elf_build_version`
   - `dwarf_build_version`
   - `ddpi_build_version`

7. `ddpi_init` initializes the `libddpi` environment.

   Before the `libddpi` functions 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 function 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 consumer objects

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

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 retrive this information. In this case, proceed to 5 on page 28.

   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 15.

---

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 function demonstrates a small subset of `libdwarf` functions that return the information to print out. More examples of DWARF APIs can be found in the `dwarfdump` utility.

Terminating the 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.

Previous to z/OS V1R6, the only method for generating debugging information was to use the `TEST` option and the debugging information was ISD only. Starting with 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 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 C/C++ compilers are currently supported.

Note: For more information about the supported compilers, see “CDA requirements and recommendations” on page 5.

The output file name is constructed using the *`object_file_name`* as the base. 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 (USS) utility that runs in the shell environment. It supports only HFS 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.
- `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.

Appendix A. Diagnosing problems

This appendix 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 APIs from the `libelf` library are called.
  - Include `libdwarf.h` and `dwarf.h` if APIs from the `libdwarf` library are called.
  - Include `libddpi.h` if APIs from the `libddpi` library are called.
- Verify that your application is compiled with the `XPLINK` compiler option if it calls APIs from the `libddpi` library.
- 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`.
- 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`, then your application may 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 Extension 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 33.

   _____

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

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 Volume 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:

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

One exception is command syntax that is published in railroad track format; screen-readable copies of z/OS books with that syntax information are separately available in HTML zipped file form upon request to compinfo@ca.ibm.com.

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 give 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, 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 OR CONDITIONS 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 may 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 illustrates programming techniques on the z/OS 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. 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 IBM's application programming interfaces.

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 or registered trademarks of International Business Machines Corporation in the United States or other countries or both:

AD/Cycle	BookManager	BookMaster
eServer	IBM	Language Environment
MVS	MVS/ESA	OS/390
S/390	z/OS	zSeries

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

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

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 on the *IBM Online Library Omnibus Edition MVS Collection*, SK2T-0710, the *z/OS Collection*, SK3T-4269, or on a tape 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 and z/OS.e Planning for Installation*, GA22-7504
- *z/OS Summary of Message and Interface Changes*, SA22-7505
- *z/OS Information Roadmap*, SA22-7500

z/OS C/C++

- *z/OS C/C++ Programming Guide*, SC09-4765
- *z/OS C/C++ User's Guide*, SC09-4767
- *z/OS C/C++ Language Reference*, SC09-4815
- *z/OS C/C++ Messages*, GC09-4819
- *z/OS C/C++ Run-Time Library Reference*, SA22-7821
- *z/OS C Curses*, SA22-7820
- *z/OS 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 Debugging Guide*, GA22-7560
- *z/OS Language Environment Programming Guide*, SA22-7561
- *z/OS Language Environment Programming Reference*, SA22-7562
- *z/OS Language Environment Writing Interlanguage Communication Applications*, SA22-7563
- *z/OS Language Environment Customization*, SA22-7564
- *z/OS Language Environment Run-Time Application Migration Guide*, GA22-7565
- *z/OS Language Environment Concepts Guide*, SA22-7567
- *z/OS Language Environment Run-Time Messages*, SA22-7566

- *z/OS Language Environment Vendor Interfaces*, SA22-7568

z/Architecture

- *z/Architecture Principles of Operations*, which is available at:
  http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/Shelves/DZ9ZBK01

INDEX

K

L

M

N

O

P

R

S

T

U

V

W

X

Program Number: 5694-A01 and 5655-G52

SC09-7653-01