Compilation Systems Volume 2 (Concepts)

0890460-050
April 1999

The information contained in this document is believed to be correct at the time of publication. It is subject to change without notice. Concurrent Computer Corporation makes no warranties, expressed or implied, concerning the information contained in this document.

To report an error or comment on a specific portion of the manual, photocopy the page in question and mark the correction or comment on the copy. Mail the copy (and any additional comments) to Concurrent Computer Corporation, 2101 W. Cypress Creek Road, Ft. Lauderdale, FL 33309–1892. Mark the envelope **"Attention: Publications Department**." This publication may not be reproduced for any other reason in any form without written permission of the publisher.

This document is based on copyrighted documentation from Novell, Inc. and is reproduced with permission.

Acknowledgment: This manual contains material contributed by 88open Consortium, Ltd. and UNIX International

In this document, the term 601 is used as an abbreviation for the phrase "PowerPC 601 RISC microprocessor." The terms 603, 604, and 620 are used similarly.

Printed in U. S. A.

Revision History:	Level:	Effective With:
Original Release -- October 1994	000	PowerUX r1.0
Previous Release -- July 1996	034	PowerUX 3.1
Current Release -- April 1999	050	PowerMAX OS 4.3

Preface

Scope of Manuals

The Compilation Systems Manual set is composed of two manuals: *Compilation Systems Volume 1 (Tools)* and *Compilation Systems Volume 2 (Concepts)*. The *Compilation Systems Volume 1 (Tools)* manual describes the features and use of several software development environment tools, analysis tools, and project-control tools. The *Compilation Systems Volume 2 (Concepts)* manual describes the concepts behind compilation systems including environments, performance analysis, and formats.

Information in this manual applies to the PowerPC™ platforms described in the *Concurrent Computer Corporation Product Catalog*.

Structure of Manuals

A brief description of the parts, chapters, and appendixes in the *Compilation Systems Volume 1 (Tools)* manual follows:

Part 1 discusses software development environment tools.

Chapter 1 introduces compilation system tools and concepts.

Chapter 2 describes the assembly language, and it discusses the assembler, **as**.

Chapter 3 summarizes the instructions, condition codes, operands, and registers associated with the PowerPC.

Chapter 4 covers the link editor, **ld**. It also discusses dynamic linking, plus the creation and use of shared objects.

Chapter 5 describes the macro processor, **m4**.

Chapter 6 presents the lexical analyzer, **lex**.

Chapter 7 presents the compiler-compiler, **yacc**.

Part 2 describes analysis tools.

Chapter 8 provides an introduction to the other chapters in this part.

Chapter 9 presents the C code browser, **cscope**.

Chapter 10 discusses the C code checker, **lint**.

Chapter 11 discusses performance analysis and use of the **analyze** and **report** utilities.

Part 3 presents project-control tools.

Chapter 12 provides an introduction to the other chapters in this part.

Chapter 13 presents the **make** utility.

Chapter 14 covers the **sccs** source code control system.

A brief description of the parts, chapters, and appendixes in the *Compilation Systems Volume 2 (Concepts)* manual follows:

Part 4 discusses environments.

Chapter 15 provides an introduction to the other chapters in this part.

Chapter 16 provides an overview of commonly-used system libraries.

Chapter 17 discusses the IEEE floating-point operations used on supporting hardware platforms.

Chapter 18 describes interfaces between C and Fortran routines on supporting hardware platforms.

Part 5 describes performance analysis concepts.

Chapter 19 provides an introduction to the other chapters in this part.

Chapter 20 provides a tutorial on program optimization, focusing on the optimizations performed by the Concurrent compilers.

Part 6 covers formats.

Chapter 21 provides an introduction to the other chapters in this part.

Chapter 22 describes the executable and linking format, ELF.

Chapter 23 discusses text description information, tdesc.

Chapter 24 describes the debugging information format, DWARF. It is primarily a reprint of the DWARF specification from UNIX International.

Chapter 25 covers the libdwarf library that provides access to DWARF debugging and line number information. It is primarily a reprint of a document from UNIX International.

Syntax Notation

The following notation is used throughout this guide:

italic	Books, reference cards, and items that the user must specify appear in *italic* type. Special terms and comments in code may also appear in *italic.*
list bold	User input appears in **list bold** type and must be entered exactly as shown. Names of directories, files, commands, options and man page references also appear in **list bold** type.

`list`	Operating system and program output such as prompts and messages and listings of files and programs appears in `list` type. Keywords also appear in `list` type.
emphasis	Words or phrases that require extra emphasis use emphasis type.
window	Keyboard sequences and window features such as push buttons, radio buttons, menu items, labels, and titles appear in window type.
`[]`	Brackets enclose command options and arguments that are optional. You do not type the brackets if you choose to specify such option or arguments.
`{}`	Braces enclose mutually exclusive choices separated by the pipe (|) character, where one choice must be selected. You do not type the braces or the pipe character with the choice.
`...`	An ellipsis follows an item that can be repeated.

The window images in this manual come from a Motif environment. If you are using another environment, your windows may differ slightly from those presented here.

Referenced Publications

The following publications are referenced in this document:

0890240	hf77 Fortran Reference Manual
0890288	HAPSE Reference Manual
0890395	NightView User's Guide
0890398	NightTrace Manual
0891019	Concurrent C Reference Manual

The vendor publications referenced in this manual may be viewed on the respective's companies WWW site.

Contents

Part 1 Software Development Environments

Chapter 1 Introduction to SDEs

Chapter 2 Assembler and Assembly Language

Chapter 3 PowerPC Instruction Set Summary

Chapter 4 Link Editor and Linking

Chapter 5 m4 Macro Processor

Chapter 6 Lexical Analysis with lex

Chapter 7 Parsing with yacc

Part 2 Analysis

Chapter 8 Introduction to Analysis

Chapter 9 Browsing Through Your Code with cscope

Chapter 10 Analyzing Your Code with lint

Chapter 11 Performance Analysis

Part 3 Project Control

Chapter 12 Introduction to Project Control

Chapter 13 Managing File Interactions with make

Chapter 14 Tracking Versions with SCCS

Chapter 17 Floating-Point Operations

Chapter 18 Inter-Language Interfacing

Part 5 Program Optimization

Chapter 19 Introduction to Program Optimization

Chapter 20 Program Optimization

Chapter 23 tdesc Information

Chapter 24 DWARF Debugging Information Format

Chapter 25 DWARF Access Library (libdwarf)

Illustrations

Screens

Tables

4
Environments

Replace with Part 4 tab

Part 4 - Environments

Part 4 Environments

15
Introduction to Environments

15
Introduction to Environments

Introduction

You can save time writing routines by calling system functions instead. You know how to write and tune your math-intensive and multi-language programs if you understand the concepts behind floating-point operations and inter-language interfacing.

This part of the manual describes implementation-dependent aspects of the environment.

Chapter 16 (“Run-Time Libraries”) categorizes, groups, and briefly describes the functions in the C, ELF, and math system libraries.

Chapter 17 (“Floating-Point Operations”) discusses IEEE single-precision and double-precision floating-point arithmetic, exception handling, operations, and implementations.

Chapter 18 (“Inter-Language Interfacing”) describes the interfaces between C and Fortran routines on supporting hardware platforms. Topics include stack frames, parameter passing, return values, register use, external names, and data types.

16
Run-Time Libraries

16
Run-Time Libraries

Introduction

PowerUX provides several system libraries which are available to software developers. This chapter introduces three of these libraries. A brief synopsis of each function in the libraries is presented. More detailed information can be found in the manual pages for the functions.

System Libraries

The following system libraries are available:

- C
- Math
- Alternate math
- ELF
- DWARF
- General-purpose

C Library

This is the basic library for C language programs. It contains functions and declarations used for file access, string testing and manipulation, character testing and manipulation, memory allocation, and other capabilities.

The following man page sections pertain to the library:

2	System Calls
3C	Standard C Library
3S	Standard I/O Library

The static C library is **`/usr/ccs/lib/libc.a`**. It is used when link editing programs which do not perform dynamic linking. Programs which do perform dynamic linking are link edited with **`/usr/ccs/lib/libc.so`**. This library contains a shared object, **`/usr/lib/libc.so.1`**, which contains the dynamic linker and other position independent functions.

Alternate C Library

An alternate static C library, **/usr/ccs/lib/libnc.a**, is available under Power UNIX. It does not support reentrancy of its functions, as does the default C library. Programs that are link edited with this alternate library will exhibit better performance. Only those programs which do not use dynamic linking and which do not depend upon the reentrancy quality of the library, however, can use this library. Programs which use dynamic linking must continue to use **/usr/ccs/lib/libc.so**. Programs which depend upon reentrancy in the library, such as programs that are link edited with the system threads library, cannot use **/usr/ccs/lib/libnc.a**. This library may be referenced during invocation of the C compiler as follows:

cc *file*.**c -lnc**

Math Library

The math library provides interfaces for commonly used mathematical functions. The functions reside in **/usr/ccs/lib/libm.a**. This library may be referenced during invocation of the C compiler as follows:

cc *file*.**c -lm**

The following man page section pertains to this library:

3M Math Library

Alternate Math Library

An alternate math library, **/usr/ccs/lib/libM.a**, is available under PowerUX. It is intended for use when the characteristics of the arguments are well-understood and higher performance is preferred to increased accuracy. This library differs from the standard math library, **/usr/ccs/lib/libm.a**, in the following ways:

- Arguments are not checked to ensure that they are valid IEEE floating-point numbers.
- Arguments are not checked for mathematical validity (for example, `sqrt(-2)`).
- For the single-precision functions, certain calculations that are performed in double precision in the standard library are performed in single precision in the alternate library. As a result, 1-bit errors can occur in some calculations.
- This alternate library uses large tables of constants as a repository of data for its calculations. Use of this library will require a larger address space than is needed with the standard library.

This alternate library may be referenced during invocation of the C compiler as follows:

cc *file*.**c -lM**

The Fortran and Ada compilation systems reference the standard math library, **`libm.a`**, by default. The C compilation system has no default math library.

ELF Library

This library provides functions that access and manipulate ELF object files. Refer to Chapter 22 ("Executable and Linking Format (ELF)") for information on ELF.

The functions reside in **`/usr/ccs/lib/libelf.a`**. This library may be referenced during invocation of the C compiler as follows:

```
cc file.c -lelf
```

The following man page section pertains to this library:

3E Executable and Linking Format Library

DWARF Library

This library provides functions that access and manipulate DWARF debugging information in ELF object files.

The functions reside in **`/usr/ccs/lib/libdwarf.a`**. This library may be referenced during invocation of the C compiler as follows:

```
cc file.c -ldwarf
```

The following man page section pertains to this library:

3DWARF Debugging with Arbitrary Record Format Library

General-Purpose Library

This library provides general-purpose functions, often maintained for compatibility with previous versions of UNIX®.

The functions reside in **`/usr/ccs/lib/libgen.a`**. This library may be referenced during invocation of the C compiler as follows:

```
cc file.c -lgen
```

The following man page section pertains to this library:

3G General-Purpose Library

Including Functions and Data

When a program is being compiled, the compilation system automatically directs the link editor to search the C library to locate and resolve references to functions and data needed by the program. For it to locate and include functions and data from other libraries, you must specify these libraries on the invocation line. For example, when using functions of

the math library, you must request that the math library be searched by including **-lm** on the invocation line:

```
cc file.c -lm
```

The **-lm** must appear after all files that reference functions in the math library. In this way, the link editor is able to use the math library to resolve references to math library functions, and thereby include these functions in the **a.out** file.

Including Declarations

To operate properly, some functions need a set of declarations. These declarations are in header files under the **/usr/include** directory. To include these header files, you must code requests in your C source program. A request is of the form:

```
#include <file.h>
```

where *file*.h is the name of the file. Because the header files define the types of functions and various preprocessor constants, they must be included before invoking the declared functions.

Listing of Functions

Input/Output Control

The input/output control functions are grouped into the following categories:

File and I/O Control and Access

Table 16-1. File and I/O Control and Access Functions

Function	Reference	Brief Description
access	**access(2)**	Determine the accessibility of a file.
basename	**basename(3G)**	Provide the last element of a path name.
chmod, fchmod	**chmod(2)**	Change the mode of a file.
chown, fchown, lchown	**chown(2)**	Change the owner and group of a file.
close	**close(2)**	Close a file descriptor.
copylist	**copylist(3G)**	Copy a file into memory.
creat	**creat(2)**	Create a new file or rewrite an existing file.
dirname	**dirname(3G)**	Provide the parent directory name of a file path name.
dup	**dup(3C)**	Duplicate an open file descriptor.
dup2	**dup2(3C)**	Duplicate an open file descriptor.
fclose	**fclose(3S)**	Close an open stream.
fcntl	**fcntl(2)**	Control an open file.
fdopen	**fopen(3S)**	Associate a file stream with an open file.
fgetpos	**fsetpos(3C)**	Get the position of a file pointer in a file stream.
fileno	**ferror(3S)**	Identify the file descriptor associated with an open stream.
filepriv	**filepriv(2)**	Control the privileges associated with a file.
flockfile	**flock(3S), flockfile(3S)**	Grant thread ownership of a file.
fopen	**fopen(3S)**	Open a file with specified permissions.
fpathconf, pathconf	**fpathconf(2)**	Get configurable path name variables.
freopen	**fopen(3S)**	Substitute a named file in place of an open file stream.
fseek	**fseek(3S)**	Reposition the file pointer in a file stream.
fsetpos	**fsetpos(3C)**	Set the position of a file pointer in a file stream.
fsync	**fsync(2)**	Synchronize a file's in-memory state with that on a physical medium.
ftrylockfile	**flock(3S), ftrylockfile(3S)**	Grant thread ownership of a file, and indicate a status of success or failure.
funlockfile	**flock(3S), funlockfile(3S)**	Relinquish file ownership granted to a thread.
getdtablesize	**getdtablesize(3C)**	Get the file descriptor table size.

Table 16-1. File and I/O Control and Access Functions (Cont.)

Function	Reference	Brief Description
`link`	**`link(2)`**	Create a new link for a file.
`lockf`	**`lockf(3C)`**	Record locking on files.
`lseek`	**`lseek(2)`**	Move a read/write file pointer.
`open`	**`open(2)`**	Open a file descriptor.
`pathfind`	**`pathfind(3G)`**	Find a named file in named directories.
`poll`	**`poll(2)`**	Multiplex I/O.
`rename`	**`rename(2)`**	Change the name of a file.
`remove`	**`remove(3C)`**	Remove a file.
`rewind`	**`fseek(3S)`**	Reposition the file pointer to the beginning of a file.
`select`	**`select(3C)`**	Perform synchronous I/O multiplexing.
`setbuf`	**`setbuf(3S)`**	Assign buffering to a file stream.
`setvbuf`	**`setbuf(3S)`**	Assign buffering to a file stream**,** but allow finer control.
`symlink`	**`symlink(2)`**	Make a symbolic link to a file.
`truncate, ftruncate`	**`truncate(3C)`**	Set a file to a specified length.
`unlink`	**`unlink(2)`**	Remove a directory entry.
`userdma`	**`userdma(2)`**	Prepare a buffer for DMA transfers.
`utime`	**`utime(2)`**	Set file access and modification times.

File and I/O Status

These functions provide status information on files and I/O operations.

Table 16-2. File and I/O Status Functions

Function	Reference	Brief Description
`clearerr`	**`ferror(3S)`**	Reset an error condition on a file stream.
`feof`	**`ferror(3S)`**	Test for end-of-file on a file stream.
`ferror`	**`ferror(3S)`**	Test for an error condition on a file stream.
`ftell`	**`fseek(3S)`**	Indicate the current position in the file.
`readlink`	**`readlink(2)`**	Read the value of a symbolic link.
`realpath`	**`realpath(3C)`**	Return a file name.
`stat, fstat, lstat`	**`stat(2)`**	Obtain file status information.

Directories

These functions support operations on directories.

Table 16-3. Directories Functions

Function	Reference	Brief Description
`alphasort`	**`scandir(3C)`**	Sort directory entries.
`chdir, fchdir`	**`chdir(2)`**	Change the working directory.
`chroot`	**`chroot(2)`**	Change the root directory.
`closedir`	**`directory(3C)`**	Close a directory.
`getdents`	**`getdents(2)`**	Read directory entries.
`mkdir`	**`mkdir(2)`**	Make a directory.
`mkdirp`	**`mkdirp(3G)`**	Create directories in a path.
`mknod`	**`mknod(2)`**	Make a directory, or a special or ordinary file.
`opendir`	**`directory(3C)`**	Open a directory.
`rmdir`	**`rmdir(2)`**	Remove a directory.
`rmdirp`	**`mkdirp(3G)`**	Remove directories in a path.
`readdir, readdir_r`	**`directory(3C)`**	Read a directory.
`rewinddir`	**`directory(3C)`**	Reset the file position to the beginning of a directory.
`scandir`	**`scandir(3C)`**	Scan a directory.
`seekdir`	**`directory(3C)`**	Seek in a directory.
`telldir`	**`directory(3C)`**	Provide a pointer to the current location in a directory.

File Systems

These functions support operations on file systems.

Table 16-4. File Systems Functions

Function	Reference	Brief Description
`mount`	**`mount(2)`**	Mount a file system.
`statvfs, fstatvfs`	**`statvfs(2)`**	Obtain file system status information.

Table 16-4. File Systems Functions (Cont.)

Function	Reference	Brief Description
`sysfs`	**`sysfs(2)`**	Obtain file system type information.
`umount`	**`umount(2)`**	Unmount a file system.
`ustat`	**`ustat(2)`**	Obtain file system statistics.

General Input

These functions support a variety of general input operations.

Table 16-5. General Input Functions

Function	Reference	Brief Description
`bgets`	**`bgets(3G)`**	Read a stream up to the next delimiter.
`fgetc`	**`getc(3S)`**	Read a character from standard input.
`fgets`	**`gets(3S)`**	Read a string from a file stream.
`fread`	**`fread(3S)`**	Read buffered data from a file stream.
`fscanf`	**`scanf(3S)`**	Read characters from a file stream.
`fwscanf`	**`fwscanf(3S)`**	Read wide characters from a file stream.
`getc`, `getc_unlocked`	**`getc(3S)`**	Read character from a file stream.
`getchar`, `getchar_unlocked`	**`getc(3S)`**	Read a character from standard input.
`gets`	**`gets(3S)`**	Read a string from standard input.
`getw`	**`getc(3S)`**	Read a word from a file stream.
`pread`	**`pread(2)`**	Perform an atomic position and read.
`read`	**`read(2)`**	Read from a file.
`scanf`	**`scanf(3S)`**	Read characters from standard input.
`sscanf`	**`scanf(3S)`**	Read characters from a string.
`swscanf`	**`fwscanf(3S)`**	Read wide characters from a string.
`ungetc`	**`ungetc(3S)`**	Put one character back on standard input.
`vfscanf`	**`vscanf(3S)`**	Read characters from a file stream by `varargs` argument list.
`vfwscanf`	**`vfwscanf(3S)`**	Read wide characters from a file stream by `varargs` argument list.
`vscanf`	**`vscanf(3S)`**	Read characters from standard input by `varargs` argument list.

Table 16-5. General Input Functions (Cont.)

Function	Reference	Brief Description
vsscanf	**vscanf(3S)**	Read characters from a string by varargs argument list.
wscanf	**vscanf(3S)**	Read characters from standard input by varargs argument list.
vswcanf	**vfwscanf(3S)**	Read wide characters from a string by varargs argument list.
vwscanf	**vfwscanf(3S)**	Read wide characters from standard input by varargs argument list.

General Output

These functions support a variety of general output operations.

Table 16-6. General Output Functions

Function	Reference	Brief Description
addsev	**addsev(3C)**	Define additional severities.
addseverity	**addseverity(3C)**	Build a list of severity levels.
fflush	**fclose(3S)**	Write all currently buffered characters to a file stream.
fmtmsg	**fmtmsg(3C)**	Display a message on standard error or the system console.
fprintf	**printf(3S)**	Write characters to a file stream.
fputc	**putc(3S)**	Write a character to standard output.
fputs	**puts(3S)**	Write a string to a file stream.
funflush	**funflush(3S)**	Discard buffered data.
fwprintf	**fwprintf(3S)**	Write wide characters to a file stream.
fwrite	**fread(3S)**	Write buffered data to a file stream.
lfmt	**lfmt(3C)**	Display an error message and pass it to logging and monitoring services.
perror	**perror(3C)**	Write an error message to standard error.
printf	**printf(3S)**	Write characters to standard output.
putc, putc_unlocked	**putc(3S)**	Write a character to standard output.
putchar, putchar_unlocked	**putc(3S)**	Write a character to standard output.
puts	**puts(3S)**	Write a string to standard output.
putw	**putc(3S)**	Write a word to a file stream.

Table 16-6. General Output Functions (Cont.)

Function	Reference	Brief Description
`pwrite`	**`pwrite(2)`**	Perform an atomic position and write.
`pfmt`	**`pfmt(3C)`**	Display an error message.
`setlabel`	**`setlabel(3C)`**	Define the label for `pfmt`.
`snprintf`	**`printf(3S)`**	Write a specified number of characters to a string.
`sprintf`	**`printf(3S)`**	Write characters to a string.
`strerror`	**`strerror(3C)`**	Write an error message to standard error.
`swprintf`	**`fwprintf(3S)`**	Write wide characters to a string.
`vfprintf`	**`vprintf(3S)`**	Write characters to a file stream by `varargs` argument list.
`vfwprintf`	**`vfwprintf(3S)`**	Write wide characters to a file stream by `varargs` argument list.
`vlfmt`	**`lfmt(3C)`**	Display an error message and pass it to logging and monitoring services, by `varargs` argument list.
`vpfmt`	**`pfmt(3C)`**	Display an error message, by `varargs` argument list.
`vprintf`	**`vprintf(3S)`**	Write characters to standard output by `varargs` argument list.
`vsprintf`	**`vprintf(3S)`**	Write characters to a string by `varargs` argument list.
`vswprintf`	**`vfwprintf(3S)`**	Write wide characters to a string by `varargs` argument list.
`vwprintf`	**`vfwprintf(3S)`**	Write wide characters to standard output by `varargs` argument list.
`wprintf`	**`fwprintf(3S)`**	Write wide characters to standard output.
`write`	**`write(2)`**	Write to a file.

Terminal I/O

These functions support terminal I/O operations.

Table 16-7. Terminal I/O Functions

Function	Reference	Brief Description
`cfgetispeed`	**`termios(2)`**	Get the input baud rate.
`cfsetispeed`	**`termios(2)`**	Set the input baud rate.
`cfgetospeed`	**`termios(2)`**	Get the output baud rate.
`cfsetospeed`	**`termios(2)`**	Set the output baud rate.
`ctermid`	**`ctermid(3S)`**	Indicate the file name for the controlling terminal.
`grantpt`	**`grantpt(3C)`**	Grant access to a slave pseudo-terminal device.

Table 16-7. Terminal I/O Functions (Cont.)

Function	Reference	Brief Description
`isatty`	**`ttyname(3C)`**	Determine if the file descriptor is associated with a terminal.
`ptsname`	**`ptsname(3C)`**	Provide the name of a slave pseudo-terminal device.
`tcdrain`	**`termios(2)`**	Wait for transmission of all output.
`tcflow`	**`termios(2)`**	Suspend transmission or reception of data.
`tcflush`	**`termios(2)`**	Discard untransmitted or unread data.
`tcgetattr`	**`termios(2)`**	Get terminal attributes.
`tcgetpgrp`	**`termios(2)`**	Get the foreground process group ID.
`tcsendbreak`	**`termios(2)`**	Send data to generate a break condition.
`tcsetattr`	**`termios(2)`**	Set terminal attributes.
`tcsetpgrp`	**`termios(2)`**	Set the foreground process group ID.
`tcsetsid`	**`termios(2)`**	Set the session ID.
`ttyname, ttyname_r`	**`ttyname(3C)`**	Provide the path name of the terminal associated with the file descriptor.
`unlockpt`	**`unclockpt(3C)`**	Unlock a pseudo-terminal master/slave pair.

STREAMS

These functions support operations on STREAMS files.

Table 16-8. STREAMS Functions

Function	Reference	Brief Description
`fattach`	**`fattach(3C)`**	Attach a STREAMS-based file descriptor to a file system object.
`fdetach`	**`fdetach(3C)`**	Detach a name from a STREAMS-based file descriptor.
`getmsg, getpmsg`	**`getmsg(2)`**	Get the next message off a stream from a STREAMS file.
`isastream`	**`isastream(3C)`**	Determine if a file descriptor represents a STREAMS file.
`putmsg, putpmsg`	**`putmsg(2)`**	Set a message to a STREAMS file.

Pipes and FIFOs

These functions support operations on pipes and FIFOs.

Table 16-9. Pipes and FIFOs Functions

Function	Reference	Brief Description
mkfifo	**mkfifo(3C)**	Create a new FIFO special file.
p2close	**p2close(3G)**	Close a pipe from a command.
p2open	**p2open(3G)**	Open a pipe to a command.
pclose	**popen(3S)**	Close a stream opened by popen.
pipe	**pipe(2)**	Create an inter-process channel
popen	**popen(3S)**	Create a pipe as a stream between the calling process and a command.

Devices

These functions support general control of devices.

Table 16-10. Devices Control Functions

Function	Reference	Brief Description
devstat, fdevstat	**devstat(2)**	Get or set device security attributes.
ioctl	**ioctl(2)**	Control a device.
major	**makedev(3C)**	Provide the major number component from a device.
makedev	**makedev(3C)**	Make a device.
minor	**makedev(3C)**	Provide the minor number component from a device.

Special Files

The special files functions support a variety of operations on special files. They are grouped into the following categories:

- "File Systems Table File" on page 16-13
- "File Systems Mount Table File" on page 16-14
- "Password File" on page 16-14
- "Shadow Password File" on page 16-15

File Systems Table File

These functions search and access information stored in the file systems table file (**/etc/vfstab**).

Table 16-11. File Systems Table File Functions

Function	Reference	Brief Description
endfsent	**getfsent(3C)**	Close **/etc/vfstab**.
getfsent	**getfsent(3C)**	Read the next line of **/etc/vfstab**.
getfsfile	**getfsent(3C)**	Read the next line of **/etc/vfstab** that matches the file system file name.
getfsspec	**getfsent(3C)**	Read the next line of **/etc/vfstab** that matches the special file name
getfstype	**getfsent(3C)**	Read the next line of **/etc/vfstab** that matches the file system type.
getvfsany	**getvfsent(3C)**	Read the next line of **/etc/vfstab** that matches the vfs table entry.
getvfsent	**getvfsent(3C)**	Read the next line of **/etc/vfstab**.
getvfsfile	**getvfsent(3C)**	Read the next line of **/etc/vfstab** that matches the file system file name.
getvfsspec	**getvfsent(3C)**	Read the next line of **/etc/vfstab** that matches the special file name
getvfstype	**getvfsent(3C)**	Read the next line of **/etc/vfstab** that matches the file system type.
setfsent	**getfsent(3C)**	Open and rewind **/etc/vfstab**.

File Systems Mount Table File

These functions search and access information stored in the file systems mount table file (**/etc/mnttab**).

Table 16-12. File Systems Mount Table File Functions

Function	Reference	Brief Description
addmntent	**getmntent(3C)**	Add a mount entry to the end of **/etc/mnttab**.
endmntent	**getmntent(3C)**	Close **/etc/mnttab**.
getmntany	**getmntent(3C)**	Read the next line of **/etc/mnttab** that matches the mount entry.
getmntent	**getmntent(3C)**	Read the next line of **/etc/mnttab**.
hasmntopt	**getmntent(3C)**	Obtain the options subfield of a mount entry that has the option.
setmntent	**getmntent(3C)**	Open and rewind **/etc/mnttab**.

Password File

These functions search and access information stored in the password file (**/etc/passwd**).

Table 16-13. Password File Functions

Function	Reference	Brief Description
endpwent	**getpwent(3G)**	Close **/etc/passwd**.
fgetpwent	**getpwent(3G)**	Read the next line of a password file.
getpw	**getpw(3G)**	Read the next line of **/etc/passwd** that matches the user id.
getpwent	**getpwent(3G)**	Read the next line of **/etc/passwd**.
putpwent	**putpwent(3C)**	Write a line to a password file.
getpwnam	**getpwent(3G)**	Read the next line of **/etc/passwd** that matches the login name.
getpwuid	**getpwent(3G)**	Read the next line of **/etc/passwd** that matches the user id.
setpwent	**getpwent(3G)**	Open and rewind **/etc/passwd**.

Shadow Password File

These functions search and access information stored in the shadow password file (**/etc/shadow**).

Table 16-14. Shadow Password File Functions

Function	Reference	Brief Description
endspent	**getspent(3G)**	Close **/etc/shadow**
fgetspent	**getspent(3G)**	Read the next line of a shadow password file.
getspent	**getspent(3G)**	Read the next line of **/etc/shadow**.
putspent	**putspent(3G)**	Write a line to a shadow password file.
getspnam	**getspent(3G)**	Read the next line of **/etc/shadow** that matches the login name.
lckpwdf	**getspent(3G)**	Obtain an exclusive lock for modification of **/etc/shadow** and **/etc/passwd**.
setspent	**getspent(3G)**	Open and rewind **/etc/shadow**.
ulckpwdf	**getspent(3G)**	Relinquish an exclusive lock for modification of **/etc/shadow** and **/etc/passwd**.

Group File

These functions search and access information stored in the group file (**/etc/group**).

Table 16-15. Group File Functions

Function	Reference	Brief Description
endgrent	**getgrent(3G)**	Close **/etc/group**.
fgetgrent	**getgrent(3G)**	Read the next line of a group file.
getgrent	**getgrent(3G)**	Read the next line of **/etc/group**.
getgrgid	**getgrent(3C)**	Read the next line of **/etc/group** that matches the group id.
getgrnam	**getgrent(3C)**	Read the next line of **/etc/group** that matches the group name.
setgrent	**getgrent(3G)**	Open and rewind **/etc/group**.

User and Accounting Information Files

These functions search and access information stored in the user information files (**/var/adm/utmp**, **/var/adm/utmpx**,**/var/adm/wtmp**, and **/var/adm/wtmpx**).

Table 16-16. User and Accounting Information Files

Function	Reference	Brief Description
endtutent	**getut(3G)**	Close **/var/adm/utmp**.
endtutxent	**getutx(3G)**	Close **/var/adm/utmpx**.
getlogin, getlogin_r	**getlogin(3C)**	Provide the login name from **/var/adm/utmp**.
getutent	**getut(3G)**	Read the next entry of **/var/adm/utmp**.
getutid	**getut(3G)**	Read the next entry of **/var/adm/utmp** that matches the id.
getutline	**getut(3G)**	Read the next entry of **/var/adm/utmp** that matches the line.
getutmp	**getutx(3G)**	Copy **utmp** fields to **utmpx** fields.
getutmpx	**getutx(3G)**	Copy **utmpx** fields to **utmp** fields.
getutxent	**getutx(3G)**	Read the next entry of **/var/adm/utmpx**.
getutxid	**getutx(3G)**	Read the next entry of **/var/adm/utmpx** that matches the id.
getutxline	**getutx(3G)**	Read the next entry of **/var/adm/utmpx** that matches the line.
pututline	**getut(3G)**	Write an entry to **/var/adm/utmp**.
pututxline	**getutx(3G)**	Write an entry to **/var/adm/utmpx**.
setutent	**getut(3G)**	Rewind **/var/adm/utmp**.
setutxent	**getutx(3G)**	Rewind **/var/adm/utmpx**.
ttyslot	**ttyslot(3C)**	Find the slot of the current user in **/var/adm/utmp**.
updwtmp	**getutx(3G)**	Update **/var/adm/wtmp** and **/var/adm/wtmpx**.
updwtmpx	**getutx(3G)**	Update **/var/adm/wtmpx** and **/var/adm/wtmp**.
utmpname	**getut(3G)**	Change the name from **/var/adm/utmp**.
utmpxname	**getutx(3G)**	Change the name from **/var/adm/utmpx**.

ELF Files

These functions access and manipulate ELF object files.

These functions use `descriptors`, which provide private handles to the various pieces of an ELF object file. A more detailed overview of the ELF files access functions is available in **`elf(3E)`**.

Table 16-17. ELF Files Functions

Function	Reference	Brief Description
`elf_begin`	**`elf_begin(3E)`**	Make a file descriptor.
`elf_cntl`	**`elf_cntl(3E)`**	Control a file descriptor.
`elf_end`	**`elf_end(3E)`**	Finish using an object file.
`elf_errmsg`	**`elf_error(3E)`**	Return an error message.
`elf_errno`	**`elf_error(3E)`**	Return an internal error number.
`elf_fill`	**`elf_fill(3E)`**	Set the fill byte.
`elf_flagdata`	**`elf_flag(3E)`**	Manipulate flags for a data descriptor.
`elf_flagehdr`	**`elf_flag(3E)`**	Manipulate flags for an ELF header descriptor.
`elf_flagelf`	**`elf_flag(3E)`**	Manipulate flags for an ELF descriptor.
`elf_flagphdr`	**`elf_flag(3E)`**	Manipulate flags for a program header descriptor.
`elf_flagscn`	**`elf_flag(3E)`**	Manipulate flags for a section descriptor.
`elf_flagshdr`	**`elf_flag(3E)`**	Manipulate flags for a section header descriptor.
`elf32_fsize`	**`elf_fsize(3E)`**	Return the size of an object file.
`elf_getarhdr`	**`elf_getarhdr(3E)`**	Retrieve an archive member header.
`elf_getarsym`	**`elf_getarsym(3E)`**	Retrieve the archive symbol table.
`elf_getbase`	**`elf_getbase(3E)`**	Get the base offset for an object file.
`elf_getdata`	**`elf_getdata(3E)`**	Get a data buffer.
`elf_newdata`	**`elf_getdata(3E)`**	Create a new data descriptor.
`elf_rawdata`	**`elf_getdata(3E)`**	Get uninterpreted bytes of a data buffer.
`elf32_getehdr`	**`elf_getehdr(3E)`**	Get an ELF header.
`elf32_newehdr`	**`elf_getehdr(3E)`**	Create an ELF header.
`elf_getident`	**`elf_getident(3E)`**	Retrieve file identification data.
`elf32_getphdr`	**`elf_getphdr(3E)`**	Get a program header.
`elf32_newphdr`	**`elf_getphdr(3E)`**	Create a program header.
`elf_getscn`	**`elf_getscn(3E)`**	Return a section descriptor.
`elf_ndxscn`	**`elf_getscn(3E)`**	Return a section table index.
`elf_newscn`	**`elf_getscn(3E)`**	Create a section.

Table 16-17. ELF Files Functions (Cont.)

Function	Reference	Brief Description
elf_nextscn	**elf_getscn(3E)**	Return a section descriptor for the next higher section.
elf32_getshdr	**elf_getshdr(3E)**	Return a section header.
elf_hash	**elf_hash(3E)**	Compute a hash value.
elf_kind	**elf_kind(3E)**	Determine the file type.
elf_next	**elf_next(3E)**	Provide sequential access to the next archive member.
elf_rand	**elf_rand(3E)**	Provide random access to an archive member.
elf_rawfile	**elf_rawfile(3E)**	Retrieve uninterpreted file contents.
elf_strptr	**elf_strptr(3E)**	Create a string pointer.
elf_update	**elf_update(3E)**	Update an ELF descriptor.
elf_version	**elf_version(3E)**	Determine **libelf**'s internal version.
elf32_xlateof	**elf_xlate(3E)**	Translate memory representations to 32-bit class file representations.
elf32_xlateom	**elf_xlate(3E)**	Translate 32-bit class file representations to memory representations.

DWARF Debugging Information

These functions access and manipulate DWARF debugging information in ELF object files.

These functions use *descriptors*, which provide private handles to the various pieces of DWARF debugging information. A more detailed overview of the DWARF debugging information access functions is available in Chapter 25.

Table 16-18. DWARF Debugging Information Functions

Function	Reference	Brief Description
dwarf_arrayorder	**dwarf_arrayorder(3DWARF)**	Return a code indicating array ordering.
dwarf_atname	**dwarf_atname(3DWARF)**	Return the attribute name of an attribute.
dwarf_attr	**dwarf_attr(3DWARF)**	Return an attribute desciptor.
dwarf_attrlist	**dwarf_attrlist(3DWARF)**	Return the number of elements in an attribute list.
dwarf_bitoffset	**dwarf_bitoffset(3DWARF)**	Return the bit offset of a bit field value.
dwarf_bitsize	**dwarf_bitsize(3DWARF)**	Return the number of bits in a bit field value.
dwarf_bytesize	**dwarf_bytesize(3DWARF)**	Return the byte size for a DIE.
dwarf_child	**dwarf_child(3DWARF)**	Identify the first child of a DIE.
dwarf_childcnt	**dwarf_childcnt(3DWARF)**	Return the number of children for a DIE.

Table 16-18. DWARF Debugging Information Functions (Cont.)

Function	Reference	Brief Description
dwarf_dealloc	**dwarf_dealloc(3DWARF)**	Free dynamic storage.
dwarf_dieline	**dwarf_dieline(3DWARF)**	Return a line number descriptor.
dwarf_diename	**dwarf_diename(3DWARF)**	Return the name for a DIE.
dwarf_dieoffset	**dwarf_dieoffset(3DWARF)**	Return the offset of a DIE.
dwarf_elemlist	**dwarf_elemlist(3DWARF)**	Return the number of an elements in an element list.
dwarf_errmsg	**dwarf_errmsg(3DWARF)**	Return an error message string.
dwarf_errno	**dwarf_errno(3DWARF)**	Return an error number.
dwarf_finish	**dwarf_finish(3DWARF)**	Release internal resources.
dwarf_formaddr	**dwarf_formaddr(3DWARF)**	Return the address value of an attribute.
dwarf_formblock	**dwarf_formblock(3DWARF)**	Return a block structure.
dwarf_formref	**dwarf_formref(3DWARF)**	Return the reference value of an attribute.
dwarf_formsdata	**dwarf_formsdata(3DWARF)**	Return the signed value of an attribute.
dwarf_formstring	**dwarf_formstring(3DWARF)**	Return the string of an attribute.
dwarf_formudata	**dwarf_formudata(3DWARF)**	Return the unsigned value of an attribute.
dwarf_fundtype	**dwarf_fundtype(3DWARF)**	Return the fundamental type of a type.
dwarf_globdie	**dwarf_globdie(3DWARF)**	Return a global DIE.
dwarf_globname	**dwarf_globname(3DWARF)**	Return the name for a global DIE.
dwarf_hasattr	**dwarf_hasattr(3DWARF)**	Indicate if a DIE has a particular attribute.
dwarf_hasform	**dwarf_hasform(3DWARF)**	Indicate if a DIE has a particular attribute form.
dwarf_hibounds	**dwarf_hibounds(3DWARF)**	Return the upper bound of an array subscript.
dwarf_highpc	**dwarf_highpc(3DWARF)**	Return the high pc for a DIE.
dwarf_init	**dwarf_init(3DWARF)**	Return a handle for accessing DWARF information.
dwarf_is1stline	**dwarf_is1stline(3DWARF)**	Indicate if a line is the first in a block.
dwarf_isbitfield	**dwarf_isbitfield(3DWARF)**	Indicate whether if a DIE represents a bit field member.
dwarf_isfundtype	**dwarf_isfundtype(3DWARF)**	Indicate whether a type represents a fundamental type.
dwarf_lineaddr	**dwarf_lineaddr(3DWARF)**	Return the address for a line number.
dwarf_lineno	**dwarf_lineno(3DWARF)**	Return the source statement line number for a line number.
dwarf_lineoff	**dwarf_lineoff(3DWARF)**	Return the offset for a line number.

Table 16-18. DWARF Debugging Information Functions (Cont.)

Function	Reference	Brief Description
dwarf_linesrc	**dwarf_linesrc(3DWARF)**	Return the name of a compilation unit for a line number.
dwarf_lobounds	**dwarf_lobounds(3DWARF)**	Return the lower bound of an array subscript.
dwarf_loclist	**dwarf_loclist(3DWARF)**	Return the number of elements in a location list.
dwarf_lowpc	**dwarf_lowpc(3DWARF)**	Return the low pc for a DIE.
dwarf_modlist	**dwarf_modlist(3DWARF)**	Return the number of elements in a type modifier list.
dwarf_nextdie	**dwarf_nextdie(3DWARF)**	Return the next DIE.
dwarf_nextglob	**dwarf_nextglob(3DWARF)**	Return the next global DIE.
dwarf_nextline	**dwarf_nextline(3DWARF)**	Return the next line number.
dwarf_nthsubscr	**dwarf_nthsubscr(3DWARF)**	Return a subscript.
dwarf_offdie	**dwarf_offdie(3DWARF)**	Return the DIE at a particular offset.
dwarf_pcfile	**dwarf_pcfile(3DWARF)**	Return the compilation unit DIE for a pc.
dwarf_pclines	**dwarf_pclines(3DWARF)**	Create a block of line numbers.
dwarf_pcscope	**dwarf_pcscope(3DWARF)**	Return the DIE for a pc scope.
dwarf_pcsubr	**dwarf_pcsubr(3DWARF)**	Return the subroutine DIE for a pc.
dwarf_prevline	**dwarf_prevline(3DWARF)**	Return the previous line number.
dwarf_seterrarg	**dwarf_seterrarg(3DWARF)**	Replace the error handler communication area.
dwarf_seterrhand	**dwarf_seterrhand(3DWARF)**	Replace the error handler.
dwarf_srclang	**dwarf_srclang(3DWARF)**	Return the source language for a compilation unit.
dwarf_srclines	**dwarf_srclines(3DWARF)**	Place all compilation unit line numbers into a block.
dwarf_stringlen	**dwarf_stringlen(3DWARF)**	Return the length of a string represented by a DIE.
dwarf_subscrcnt	**dwarf_subscrcnt(3DWARF)**	Return the number of subscript attributes for a type.
dwarf_subscrtype	**dwarf_subscrtype(3DWARF)**	Return the type of a subscript element.
dwarf_tag	**dwarf_tag(3DWARF)**	Return the tag for a DIE.
dwarf_typeof	**dwarf_typeof(3DWARF)**	Return a type descriptor for a type.
dwarf_udtype	**dwarf_udtype(3DWARF)**	Return a DIE for a user defined type.
dwarf_is1stline	**dwarf_is1stline(3DWARF)**	Indicate if a line is the first in a block.
dwarf_isbitfield	**dwarf_isbitfield(3DWARF)**	Indicate whether if a DIE represents a bit field member.

Table 16-18. DWARF Debugging Information Functions (Cont.)

Function	Reference	Brief Description
dwarf_isfundtype	**dwarf_isfundtype(3DWARF)**	Indicate whether a type represents a fundamental type.
dwarf_lineaddr	**dwarf_lineaddr(3DWARF)**	Return the address for a line number.
dwarf_lineno	**dwarf_lineno(3DWARF)**	Return the source statement line number for a line number.
dwarf_lineoff	**dwarf_lineoff(3DWARF)**	Return the offset for a line number.
dwarf_linesrc	**dwarf_linesrc(3DWARF)**	Return the name of a compilation unit for a line number.
dwarf_lobounds	**dwarf_lobounds(3DWARF)**	Return the lower bound of an array subscript.
dwarf_loclist	**dwarf_loclist(3DWARF)**	Return the number of elements in a location list.
dwarf_lowpc	**dwarf_lowpc(3DWARF)**	Return the low pc for a DIE.
dwarf_modlist	**dwarf_modlist(3DWARF)**	Return the number of elements in a type modifier list.
dwarf_nextdie	**dwarf_nextdie(3DWARF)**	Return the next DIE.
dwarf_nextglob	**dwarf_nextglob(3DWARF)**	Return the next global DIE.
dwarf_nextline	**dwarf_nextline(3DWARF)**	Return the next line number.
dwarf_nthsubscr	**dwarf_nthsubscr(3DWARF)**	Return a subscript.
dwarf_offdie	**dwarf_offdie(3DWARF)**	Return the DIE at a particular offset.
dwarf_pcfile	**dwarf_pcfile(3DWARF)**	Return the compilation unit DIE for a pc.
dwarf_pclines	**dwarf_pclines(3DWARF)**	Create a block of line numbers.
dwarf_pcscope	**dwarf_pcscope(3DWARF)**	Return the DIE for a pc scope.
dwarf_pcsubr	**dwarf_pcsubr(3DWARF)**	Return the subroutine DIE for a pc.
dwarf_prevline	**dwarf_prevline(3DWARF)**	Return the previous line number.
dwarf_seterrarg	**dwarf_seterrarg(3DWARF)**	Replace the error handler communication area.
dwarf_seterrhand	**dwarf_seterrhand(3DWARF)**	Replace the error handler.
dwarf_srclang	**dwarf_srclang(3DWARF)**	Return the source language for a compilation unit.
dwarf_srclines	**dwarf_srclines(3DWARF)**	Place all compilation unit line numbers into a block.
dwarf_stringlen	**dwarf_stringlen(3DWARF)**	Return the length of a string represented by a DIE.
dwarf_subscrcnt	**dwarf_subscrcnt(3DWARF)**	Return the number of subscript attributes for a type.
dwarf_subscrtype	**dwarf_subscrtype(3DWARF)**	Return the type of a subscript element.

Table 16-18. DWARF Debugging Information Functions (Cont.)

Function	Reference	Brief Description
dwarf_tag	**dwarf_tag(3DWARF)**	Return the tag for a DIE.
dwarf_typeof	**dwarf_typeof(3DWARF)**	Return a type descriptor for a type.
dwarf_udtype	**dwarf_udtype(3DWARF)**	Return a DIE for a user defined type.

Shared Objects

These functions support control of shared objects.

Table 16-19. Shared Objects Functions

Function	Reference	Brief Description
dlclose	**dlclose(3C)**	Close a shared object.
dlerror	**dlerror(3C)**	Obtain diagnostic information..
dlopen	**dlopen(3C)**	Open a shared object.
dlsym	**dlsym(3C)**	Obtain the address of a symbol in a shared object.

Temporary Files

These functions support control of temporary files.

Table 16-20. Temporary Files

Function	Reference	Brief Description
mktemp	**mktemp(3C)**	Create file name using a template.
tempnam	**tmpnam(3S)**	Create a temporary file name.
tmpfile	**tmpfile(3S)**	Create a temporary file.
tmpnam	**tmpnam(3S)**	Create a temporary file name.

Strings and Characters

These functions provide operations on characters and strings of characters. They are grouped into the following categories:

- "String Manipulation" on page 16-23
- "Wide String Manipulation" on page 16-24

String Manipulation

These functions manipulate character strings.

Table 16-21. String Manipulation Functions

Function	Reference	Brief Description
`confstr`	**`confstr(3C)`**	Obtain a configurable string.
`index`	**`string(3C)`**	Locate the first occurrence of a character in a string.
`rindex`	**`string(3C)`**	Locate the last occurrence of a character in a string.
`strcadd`	**`strccpy(3G)`**	Copy a string, compressing escape codes, and point to the terminating null byte.
`strcat`	**`string(3C)`**	Concatenate two strings.
`strccpy`	**`strccpy(3G)`**	Copy a string, compressing escape codes.
`strchr`	**`string(3C)`**	Search a string for character.
`strcmp`	**`string(3C)`**	Compare two strings.
`strcoll`	**`strcoll(3C)`**	Sort strings using locale-specific collation tables.
`strcpy`	**`string(3C)`**	Copy a string.
`strcspn`	**`string(3C)`**	Obtain the length of the initial string not containing a set of characters.
`strdup`	**`string(3C)`**	Obtain a pointer to a new string.
`streadd`	**`strccpy(3G)`**	Copy a string, expanding escape codes, and point to the terminating null byte.
`strecpy`	**`strccpy(3G)`**	Copy a string, expanding escape codes.
`strfind`	**`str(3G)`**	Locate the first occurrence of a string.
`strlen`	**`string(3C)`**	Obtain the length of a string.
`strncat`	**`string(3C)`**	Concatenate two strings, with a maximum length.
`strncmp`	**`string(3C)`**	Compare two strings, with a maximum length.
`strncpy`	**`string(3C)`**	Copy a string, with a maximum length.
`strpbrk`	**`string(3C)`**	Search a string for a set of characters.
`strrchr`	**`string(3C)`**	Search a string backwards for a character.

Table 16-21. String Manipulation Functions (Cont.)

Function	Reference	Brief Description
`strrspn`	**`str(3G)`**	Locate the first character to be trimmed.
`strspn`	**`string(3C)`**	Obtain the length of the initial string containing a set of characters.
`strstr`	**`string(3C)`**	Locate the first occurrence of a substring in a string.
`strtok, strtok_r`	**`string(3C)`**	Search a string for a token separated by any of a set of characters.
`strtrns`	**`str(3G)`**	Transform a string.
`strxfrm`	**`strxfrm(3C)`**	Transform a string.

Wide String Manipulation

These functions manipulate wide character strings.

Table 16-22. Wide String Manipulation Functions

Function	Reference	Brief Description
`wcscat`	**`wcscat(3C)`**	Concatenate two wide character strings.
`wcschr`	**`wcschr(3C)`**	Scan a wide character string.
`wcscmp`	**`wcscmp(3C)`**	Compare two wide character strings.
`wcscoll`	**`wcscoll(3C)`**	Compare two wide character strings using collating information.
`wcscpy`	**`wcscpy(3C)`**	Copy a wide character string.
`wcscspn`	**`wcscspn(3C)`**	Obtain the length of a complementary wide character substring.
`wcsftime`	**`wcsftime(3C)`**	Convert a date and time to a wide character string.
`wcslen`	**`wcslen(3C)`**	Obtain the length of a wide character string.
`wcsncat`	**`wcsncat(3C)`**	Concatenate two wide character strings, with bound.
`wcsncmp`	**`wcsncmp(3C)`**	Compare two wide character strings, with bound.
`wcsncpy`	**`wcsncpy(3C)`**	Copy a wide character string, with bound.
`wcspbrk`	**`wcspbrk(3C)`**	Scan a wide character string for wide characters.
`wcsrchr`	**`wcsrchr(3C)`**	Reverse the scan of a wide character string for wide characters.
`wcsspn`	**`wcsspn(3C)`**	Obtain the length of a wide character substring.
`wcsstr`	**`wcsstr(3C)`**	Find a wide character substring in a wide character string.
`wcstod`	**`wcstod(3C)`**	Convert a wide character string to a double-precision value.
`wcstof`	**`wcstof(3C)`**	Convert a wide character string to a single-precision value.
`wcstok`	**`wcstok(3C)`**	Split a wide character string into tokens.
`wcstold`	**`wcstod(3C)`**	Convert a wide character string to a long double-precision value.

Table 16-22. Wide String Manipulation Functions (Cont.)

Function	Reference	Brief Description
`wcstol`	**`wcsstrtol(3C)`**	Convert a wide character string to a long integer value.
`wcstoul`	**`wcsstrtol(3C)`**	Convert a wide character string to an unsigned long integer value.
`wcswidth`	**`wcswidth(3C)`**	Determine the number of column positions for a wide character string.
`wcsxfrm`	**`wcsxfrm(3C)`**	Transform a wide character string.
`wctob`	**`wctob(3C)`**	Provide the single byte representation of a wide character.

Character Test

These functions test characters.

Table 16-23. Character Test Functions

Function	Reference	Brief Description
`isalnum`	**`ctype(3C)`**	Determine if the character is an alphanumeric character.
`isalpha`	**`ctype(3C)`**	Determine if the character is an alphabetic character.
`isascii`	**`ctype(3C)`**	Determine if the character is an ASCII character.
`iscntrl`	**`ctype(3C)`**	Determine if the character is a control character.
`isdigit`	**`ctype(3C)`**	Determine if the character is a digit.
`isgraph`	**`ctype(3C)`**	Determine if the character is a printable character.
`islower`	**`ctype(3C)`**	Determine if the character is a lowercase letter.
`isprint`	**`ctype(3C)`**	Determine if the character is a printing character.
`ispunct`	**`ctype(3C)`**	Determine if the character is a punctuation character.
`isspace`	**`ctype(3C)`**	Determine if the character is a white space character.
`isupper`	**`ctype(3C)`**	Determine if the character is an uppercase letter.
`isxdigit`	**`ctype(3C)`**	Determine if the character is a hex digit.

Wide Character Test

These functions test wide characters.

Table 16-24. Wide Character Test Functions

Function	Reference	Brief Description
`iswalnum`	**`wctype(3C)`**	Determine if the wide character is an alphanumeric character.
`iswalpha`	**`wctype(3C)`**	Determine if the wide character is an alphabetic character.
`iswcntrl`	**`wctype(3C)`**	Determine if the wide character is a control character.
`iswctype`	**`iswctype(3C)`**	Determines if the wide character is of a particular wide character class.
`iswdigit`	**`wctype(3C)`**	Determine if the wide character is a digit.
`iswgraph`	**`wctype(3C)`**	Determine if the wide character is a printable character.
`iswlower`	**`wctype(3C)`**	Determine if the wide character is a lowercase letter.
`iswprint`	**`wctype(3C)`**	Determine if the wide character is a printing character.
`iswpunct`	**`wctype(3C)`**	Determine if the wide character is a punctuation character.
`iswspace`	**`wctype(3C)`**	Determine if the wide character is a white space character.
`iswupper`	**`wctype(3C)`**	Determine if the wide character is an uppercase letter.
`iswxdigit`	**`wctype(3C)`**	Determine if the wide character is a hex digit.
`wcwidth`	**`wcwidth(3C)`**	Determine the number of column positions for a wide character.
`wcswidth`	**`wcswidth(3C)`**	Determine the number of column positions for a wide character string.

Character Translation

These functions translate characters and character strings.

Table 16-25. Character Translation Functions

Function	Reference	Brief Description
`iconv`	**`iconv(3C)`**	Convert characters from one code set to another.
`iconv_close`	**`iconv_close(3C)`**	Close a code set conversion file descriptor.
`iconv_open`	**`iconv_open(3C)`**	Open a code set conversion file descriptor.
`toascii`	**`conv(3C)`**	Convert an integer value to ASCII character.
`tolower, _tolower`	**`conv(3C)`**	Convert character to lowercase.
`toupper, _toupper`	**`conv(3C)`**	Convert character to uppercase.

Multibyte and Wide Characters

These functions support operations on multibyte and wide characters.

Table 16-26. Multibyte and Wide Characters Functions

Function	Reference	Brief Description
`mblen`	**`mbchar(3C)`**	Determine the number of bytes in a multibyte character.
`mbrlen`	**`mbchar(3C)`**	Determine the number of bytes in a multibyte character, using a conversion state.
`mbrtowc`	**`mbchar(3C)`**	Convert a multibyte character to a wide character, using a conversion state.
`mbsrtowcs`	**`mbstring(3C)`**	Convert a multibyte character string to a wide character string, using a conversion state
`mbstowcs`	**`mbstring(3C)`**	Convert a multibyte character string to a wide character string.
`mbtowc`	**`mbchar(3C)`**	Convert a multibyte character to a wide character.
`sisinit`	**`sisinit(3C)`**	Test for an initial multibyte conversion state.
`wcrtomb`	**`mbchar(3C)`**	Convert a wide character to a multibyte character, using a conversion state.
`wcsrtombs`	**`mbstring(3C)`**	Convert a wide character string to a multibyte character string, using a conversion state.
`wcstombs`	**`mbstring(3C)`**	Convert a wide character string to a multibyte character string.

Regular Expression and Pattern Matching

These functions support operations involving regular expressions and patterns.

Table 16-27. Regular Expression and Pattern Matching Functions

Function	Reference	Brief Description
`advance`	**`regexpr(3G)`**	Step and perform a restricted comparison with a regular expression.
`bufsplit`	**`bufsplit(3G)`**	Split a buffer into fields.
`compile`	**`regexpr(3G)`**	Compile a regular expression.
`fnmatch`	**`fnmatch(3C)`**	Match a file name or pattern.
`glob`	**`glob(3C)`**	Generate a path name matching a pattern.
`globfree`	**`glob(3C)`**	Free space allocated in a previous call to `glob`.
`gmatch`	**`gmatch(3G)`**	Perform shell global pattern matching.
`regcmp`	**`regcmp(3G)`**	Compile a regular expression.
`regcomp`	**`regcomp(3C)`**	Compile a regular expression.

Table 16-27. Regular Expression and Pattern Matching Functions (Cont.)

Function	Reference	Brief Description
`regerror`	**`regcomp(3C)`**	Provide a printable error string.
`regex`	**`regcmp(3G)`**	Execute a compiled regular expression.
`regexec`	**`regcomp(3C)`**	Compare with a regular expression.
`regfree`	**`regcomp(3C)`**	Free space allocated in a previous call to `regcomp`.
`step`	**`regexpr(3G)`**	Step and compare with a regular expression.
`wordexp`	**`wordexp(3C)`**	Perform word expansions.
`wordfree`	**`wordexp(3C)`**	Free space allocated in a previous call to `wordexp`.

Memory

These functions provide operations on blocks of memory. They are grouped into the following categories:

- "Memory Manipulation" on page 16-28
- "Memory Allocation" on page 16-29
- "Memory Control" on page 16-30
- "Shared Memory" on page 16-30

Memory Manipulation

These functions locate characters in a memory area and copy characters from one memory area to another.

Table 16-28. Memory Manipulation Functions

Function	Reference	Brief Description
`bcmp`	**`bstring(3C)`**	Compare two blocks of memory.
`bcopy`	**`bstring(3C)`**	Copy a block of memory.
`bzero`	**`bstring(3C)`**	Zero out a block of memory.
`ffs`	**`ffs(3C)`**	Find the first set bit in a value.
`memccpy`	**`memory(3C)`**	Copy characters from one memory area to another until a given character is found.
`memchr`	**`memory(3C)`**	Obtain a pointer to the first occurrence of a given character in a memory area.
`memcmp`	**`memory(3C)`**	Compare two memory areas.

Table 16-28. Memory Manipulation Functions (Cont.)

Function	Reference	Brief Description
`memcpy`	**`memory(3C)`**	Copy characters from one memory area to another.
`memset`	**`memory(3C)`**	Set the first characters in a memory area to a character value.
`memmove`	**`memory(3C)`**	Copy characters from one memory area to another until a given character is found.
`swab`	**`swab(3C)`**	Swap bytes.

Memory Allocation

These functions provide a means by which memory can be dynamically allocated or freed.

Table 16-29. Memory Allocation Functions

Function	Reference	Brief Description
`brk, sbrk`	**`brk(2)`**	Change the data segment space allocation.
`calloc`	**`malloc(3C)`**	Allocate an area of zeroed storage.
`free`	**`malloc(3C)`**	Free some previously allocated storage.
`mallinfo`	**`malloc(3C)`**	Provide information describing the usage of allocated storage.
`malloc`	**`malloc(3C)`**	Allocate storage.
`memalign`	**`malloc(3C)`**	Allocate storage on a specific byte-aligned boundary.
`realloc`	**`malloc(3C)`**	Change the size of allocated storage.
`valloc`	**`malloc(3C)`**	Allocate storage on a page-aligned boundary.

Memory Control

These functions control pages in memory.

Table 16-30. Memory Control Functions

Function	Reference	Brief Description
`memcntl`	**`memcntl(2)`**	Control operations over the address space.
`mincore`	**`mincore(2)`**	Determine the residency of memory pages.
`mlock`	**`mlock(3C)`**	Lock pages in memory.
`mlockall`	**`mlockall(3C)`**	Lock an address space in memory.
`mmap`	**`mmap(2)`**	Map pages of memory.
`munmap`	**`munmap(2)`**	Unmap pages of memory.
`mprotect`	**`mprotect(2)`**	Set the protection of memory mapping.
`msync`	**`msync(3C)`**	Synchronize memory with physical storage.
`munlock`	**`mlock(3C)`**	Unlock pages in memory.
`munlockall`	**`mlockall(3C)`**	Unlock an address space in memory.
`plock`	**`plock(2)`**	Lock segments into memory, or unlock text or data segments.

Shared Memory

These functions support operations on shared memory.

Table 16-31. Shared Memory Control Functions

Function	Reference	Brief Description
`shmat`	**`shmop(2)`**	Attach the shared memory segment to the data segment of the calling process.
`shmbind`	**`shmbind(2)`**	Bind a shared memory segment to a physical address.
`shmctl`	**`shmctl(2)`**	Perform shared memory control operations.
`shmdt`	**`shmop(2)`**	Detach the shared memory segment from the data segment of the calling process.
`shmget`	**`shmget(2)`**	Get a shared memory segment identifier.

Data Structures

These functions provide operations on tables, trees, and queues. They are grouped into the following categories:

Tables

These functions manage tables. Because none of these functions allocate storage, sufficient memory must be allocated before using them.

Table 16-32. Tables Functions

Function	Reference	Brief Description
`bsearch`	**`bsearch(3C)`**	Search a table using a binary search.
`lfind`	**`lsearch(3C)`**	Find an element in a library tree.
`lsearch`	**`lsearch(3C)`**	Look for and add an element in a binary tree.
`qsort`	**`qsort(3C)`**	Sort a table using the quick-sort algorithm.

Hash Tables

These functions manage hash search tables.

Table 16-33. Hash Tables Functions

Function	Reference	Brief Description
`hcreate`	**`hsearch(3C)`**	Create a hash table.
`hdestroy`	**`hsearch(3C)`**	Destroy a hash table.
`hsearch`	**`hsearch(3C)`**	Search a hash table.

File Trees

These functions traverse file trees.

Table 16-34. File Trees Functions

Function	Reference	Brief Description
`ftw`	**`ftw(3C)`**	Walk a file tree.
`nftw`	**`ftw(3C)`**	Walk a file tree in an enhanced mode.

Binary Trees

These functions manage binary trees.

Table 16-35. Binary Trees Functions

Function	Reference	Brief Description
`tdelete`	**`tsearch(3C)`**	Delete nodes from a binary tree.
`tfind`	**`tsearch(3C)`**	Find an element in a binary tree.
`tsearch`	**`tsearch(3C)`**	Look for and add an element to a binary tree.
`twalk`	**`tsearch(3C)`**	Walk through a binary tree.

Message Queues

These functions support operations on message queues.

Table 16-36. Message Queues Functions

Function	Reference	Brief Description
`mq_close`	**`mq_close(3)`**	Close a message queue.
`mq_getattr`	**`mq_getattr(3)`**	Get attributes of a message queue.
`mq_notify`	**`mq_notify(3)`**	Attach notification request to a message queue.
`mq_open`	**`mq_open(3)`**	Open a message queue.
`mq_receive`	**`mq_receive(3)`**	Receive a message from a message queue.
`mq_send`	**`mq_send(3)`**	Send a message to a message queue.
`mq_setattr`	**`mq_setattr(3)`**	Set attributes of a message queue.
`mq_unlink`	**`mq_unlink(3)`**	Unlink a message queue.
`msgctl`	**`msgctl(2)`**	Control message operations.

Table 16-36. Message Queues Functions (Cont.)

Function	Reference	Brief Description
`msgget`	**`msgget(2)`**	Get a message queue identifier.
`msgrcv`	**`msgop(2)`**	Receive a message.
`msgsnd`	**`msgop(2)`**	Send a message.

Queues

These functions manipulate queues built from doubly linked lists.

Table 16-37. Queues Functions

Function	Reference	Brief Description
`insque`	**`insque(3C)`**	Insert element into a queue.
`remque`	**`insque(3C)`**	Delete element from a queue.

Semaphores

These functions support operations on semaphores.

Table 16-38. Semaphores Functions

Function	Reference	Brief Description
`semctl`	**`semctl(2)`**	Control semaphores.
`semget`	**`semget(2)`**	Get a set of semaphores.
`semop`	**`semop(2)`**	Atomically perform semaphore operations.

Date and Time

These functions access and reformat the current date and time, access the POSIX timer, and access the interval timer. They are grouped into the following categories:

- "General Date and Time" on page 16-34
- "Interval Timer" on page 16-35
- "POSIX Timer" on page 16-35

General Date and Time

These functions access and manipulate the current date and time.

Table 16-39. General Date and Time Functions

Function	Reference	Brief Description
`adjtime`	**`adjtime(2)`**	Correct the time to allow synchronization of the system clock.
`asctime, asctime_r`	**`ctime(3C)`**	Return the string representation of the date and time.
`ascftime`	**`strftime(3C)`**	Return the string representation of the date and time based on a format string.
`ctime, ctime_r`	**`ctime(3C)`**	Return the string representation of the date and time, given an integer form.
`cftime`	**`strftime(3C)`**	Return the string representation of the date and time based on a format string, given an integer form.
`clock`	**`clock(3C)`**	Report the CPU time used.
`difftime`	**`difftime(3C)`**	Compute the difference between two calendar times.
`getdate`	**`getdate(3C)`**	Convert a user-defined date and/or time specification.
`gettimeofday`	**`gettimeofday(3C)`**	Get the system's current time.
`gmtime, gmtime_r`	**`ctime(3C)`**	Return the Greenwich mean time.
`localtime, localtime_r`	**`ctime(3C)`**	Return the local time.
`mktime`	**`mktime(3C)`**	Convert a time to a calendar time.
`settime`	**`stime(2)`**	Set the system's time and date.
`settimeofday`	**`settimeofday(3C)`**	Set the system's current time.
`strftime`	**`strftime(3C)`**	Convert a date and time to a string.
`strptime`	**`strtime(3C)`**	Convert a string to a date and time.
`time`	**`time(2)`**	Obtain the time since UTC.
`times`	**`times(2)`**	Obtain process and child process times.
`tzset`	**`ctime(3C)`**	Set the time zone field from an environment variable.

Interval Timer

These functions access the interval timer.

Table 16-40. Interval Timer Functions

Function	Reference	Brief Description
`getitimer`	**`getitimer(3C)`**	Get the value of the interval timer.
`setitimer`	**`getitimer(3C)`**	Set the value of the interval timer.

POSIX Timer

These functions access the POSIX clock and per-process timer.

Table 16-41. POSIX Timer Functions

Function	Reference	Brief Description
`clock_getres`	**`clock_getres(3C)`**	Get the resolution of the POSIX clock.
`clock_gettime`	**`clock_gettime(3C)`**	Get the value of the POSIX clock.
`clock_settime`	**`clock_settime(3C)`**	Set the value of the POSIX clock.
`posix_clocks`	**`posix_clocks(2)`**	Get or set a POSIX clock.
`posix_timers`	**`posix_timers(2)`**	Support the per-process POSIX timers.
`timer_create`	**`timer_create(3C)`**	Create a POSIX per-process timer.
`timer_delete`	**`timer_delete(3C)`**	Delete a POSIX per-process timer.
`timer_getoverrun`	**`timer_getoverrun(3C)`**	Get the overrun count for a POSIX per-process timer.
`timer_gettime`	**`timer_gettime(3C)`**	Get the value of a POSIX per-process timer.
`timer_settime`	**`timer_settime(3C)`**	Arm a POSIX per-process timer.

Internationalization

The functions support the internationalization of data and messages. They are grouped according to the following categories:

- “Locales” on page 16-36
- “Message Catalogs” on page 16-36

Locales

These functions support the use of locales.

Table 16-42. Locales Functions

Function	Reference	Brief Description
`nl_langinfo`	**`nl_langinfo(3C)`**	Obtain locale-specific information.
`setlocale`	**`setlocale(3C)`**	Establish the current locale name.
`localeconv`	**`localeconv(3C)`**	Obtain numeric and monetary formatting information.
`strfmon`	**`strfmon(3C)`**	Convert a monetary value to a string.

Message Catalogs

These functions support the use of message catalogs.

Table 16-43. Message Catalogs Functions

Function	Reference	Brief Description
`catopen`	**`catopen(3C)`**	Open a message catalog.
`catclose`	**`catopen(3C)`**	Close a message catalog.
`catgets`	**`catgets(3C)`**	Read a message from a message catalog.
`gettxt`	**`gettxt(3C)`**	Read a text string from a message catalog.
`setcat`	**`setcat(3C)`**	Define the default message catalog.

Mathematic and Numeric

The functions provide mathematical, arithmetic, and numeric operations, as well as control over the floating-point environment They are grouped according to the following categories:

- "Pseudo-Random Number Generation Functions" on page 16-42

Trigonometric

These functions are used to compute angles (in radian measure), sines, cosines, and tangents.

Table 16-44. Trigonometric Functions

Function	Reference	Brief Description
`acos, acosf`	**`trig(3M)`**	Arc cosine.
`asin, asinf`	**`trig(3M)`**	Arc sine.
`atan, atanf`	**`trig(3M)`**	Arc tangent.
`atan2, atan2f`	**`trig(3M)`**	Arc tangent of a ratio.
`cos, cosf`	**`trig(3M)`**	Cosine.
`sin, sinf`	**`trig(3M)`**	Sine.
`tan, tanf`	**`trig(3M)`**	Tangent.

Bessel

These functions are used to calculate bessel functions of the first and second kinds of several orders.

Table 16-45. Bessel Functions

Function	Reference	Brief Description
`j0`	**`bessel(3M)`**	Bessel function of the first kind of order 0.
`j1`	**`bessel(3M)`**	Bessel function of the first kind of order 1.
`jn`	**`bessel(3M)`**	Bessel function of the first kind or order n.
`y0`	**`bessel(3M)`**	Bessel function of the second kind of order 0.
`y1`	**`bessel(3M)`**	Bessel function of the second kind of order 1.
`yn`	**`bessel(3M)`**	Bessel function of the second kind of order n.

Hyperbolic

These functions are used to compute the hyperbolic sine, cosine, and tangent.

Table 16-46. Hyperbolic Functions

Function	Reference	Brief Description
`acosh`	**`sinh(3M)`**	Inverse hyperbolic cosine.
`asinh`	**`sinh(3M)`**	Inverse hyperbolic sine.
`atanh`	**`sinh(3M)`**	Inverse hyperbolic tangent.
`cosh, coshf`	**`sinh(3M)`**	Hyperbolic cosine.
`sinh, sinhf`	**`sinh(3M)`**	Hyperbolic sine.
`tanh, tanhf`	**`sinh(3M)`**	Hyperbolic tangent.

Miscellaneous Mathematic Functions

These functions cover a wide variety of operations, such as natural logarithm, exponential, and absolute value. In addition, several functions are provided to truncate the integer portion of floating-point values.

Table 16-47. Miscellaneous Mathematical Functions

Function	Reference	Brief Description
`ceil, ceilf`	**`floor(3M)`**	Smallest integral value not less than a given value.
`cbrt`	**`exp(3M)`**	Cube root.
`erf`	**`erf(3M)`**	Error function.
`erfc`	**`erf(3M)`**	Complementary error function.
`copysign`	**`floor(3M)`**	Copy of given value with a given sign.
`exp, expf`	**`exp(3M)`**	Exponential (base e).
`expm1`	**`exp(3M)`**	Equivalent to exp(x)-1.0.
`fabs, fabsf`	**`floor(3M)`**	Absolute value.
`floor, floorf`	**`floor(3M)`**	Largest integral value not greater than a given value.
`fmod, fmodf`	**`floor(3M)`**	Remainder of division of two given values.

Table 16-47. Miscellaneous Mathematical Functions (Cont.)

Function	Reference	Brief Description
`gamma, lgamma`	**`gamma(3M)`**	Natural logarithm of the absolute value of the result of applying the gamma function to a given value.
`hypot`	**`hypot(3M)`**	Square root of the sum of the squares of two values.
`log, logf`	**`exp(3M)`**	Natural logarithm.
`log10, log10f`	**`exp(3M)`**	Logarithm base ten.
`log1p`	**`exp(3M)`**	Equivalent to log(1.0+x).
`matherr`	**`matherr(3M)`**	Error-handling function for math functions.
`pow, powf`	**`exp(3M)`**	A given value raised to another given value.
`remainder`	**`floor(3M)`**	Remainder of division of two given values.
`rint`	**`floor(3M)`**	Nearest integral value to a given floating-point value.
`sqrt, sqrtf`	**`exp(3M)`**	Square root.

Numeric Conversion

These functions perform numeric conversions.

Table 16-48. Numeric Conversion Functions

Function	Reference	Brief Description
`a64l`	**`a64l(3C)`**	Convert a base-64 ASCII string to a long integer value.
`abs`	**`abs(3C)`**	Obtain the absolute integer value.
`atof`	**`strtod(3C)`**	Convert a string to a single-precision value.
`atoi`	**`strtol(3C)`**	Convert a string to an integer value.
`atol`	**`strtol(3C)`**	Convert a string to a long integer value.
`ecvt`	**`ecvt(3C)`**	Convert a double-precision value to a string.
`ecvtl`	**`ecvt(3C)`**	Convert a long double-precision value to a string.
`fcvt`	**`ecvt(3C)`**	Convert a double-precision value to a string using Fortran format.
`fcvtl`	**`ecvt(3C)`**	Convert a long double-precision value to a string using Fortran format.

Table 16-48. Numeric Conversion Functions (Cont.)

Function	Reference	Brief Description
`frexp`	**`frexp(3C)`**	Split a double-precision value into mantissa and exponent.
`frexpl`	**`frexp(3C)`**	Split a long double-precision value into mantissa and exponent.
`gcvt`	**`ecvt(3C)`**	Convert a double-precision value to a string in the style of Fortran F or E format.
`gcvtl`	**`ecvt(3C)`**	Convert a long double-precision value to a string in the style of Fortran F or E format.
`labs`	**`abs(3C)`**	Return the absolute integer value.
`ldexp`	**`frexp(3C)`**	Combine the mantissa and the exponent of a double-precision value.
`ldexpl`	**`frexp(3C)`**	Combine the mantissa and the exponent of a long double-precision value.
`logb`	**`frexp(3C)`**	Obtain the radix exponent of a double-precision value.
`logbl`	**`frexp(3C)`**	Obtain the radix exponent of long double-precision value.
`ltol3`	**`l3tol(3C)`**	Convert long integer values to 3-byte integer values.
`l3tol`	**`l3tol(3C)`**	Convert 3-byte integer values to long integer values
`l64a`, `l64a_r`	**`a64l(3C)`**	Convert a long integer value to a base-64 ASCII string.
`modf`	**`frexp(3C)`**	Split the mantissa of a double-precision value into integer and fraction parts.
`modff`	**`frexp(3C)`**	Split the mantissa of long double-precision value into integer and fraction parts.
`modfl`	**`frexp(3C)`**	Split mantissa of a single-precision value into integer and fraction parts.
`nextafter`	**`frexp(3C)`**	Return the next representable double-precision value.
`nextafterl`	**`frexp(3C)`**	Return the next representable long double-precision value.
`scalb`	**`frexp(3C)`**	Perform radix scaling for a double-precision value.
`scalbl`	**`frexp(3C)`**	Perform radix scaling for a long double-precision value.
`strtod`	**`strtod(3C)`**	Convert a string to a double-precision value.

Table 16-48. Numeric Conversion Functions (Cont.)

Function	Reference	Brief Description
`strtold`	**`strtod(3C)`**	Convert a string to a long double-precision value.
`strtol`	**`strtol(3C)`**	Convert a string to a long integer value.
`strtoul`	**`strtol(3C)`**	Convert a string to an unsigned long integer value.

Other Arithmetic

These functions provide simple arithmetic operations.

Table 16-49. Other Arithmetic Functions

Function	Reference	Brief Description
`div`	**`div(3C)`**	Divide two integers.
`ldiv`	**`div(3C)`**	Divide two long integers.

Floating-Point Environment

These functions provide control over the IEEE floating-point environment used by the program.

Table 16-50. Floating-Point Environment Functions

Function	Reference	Brief Description
`finite`, `finitel`	**`isnan(3C)`**	Determine if the number is neither infinity nor a NaN.
`fpclass`, `fpclassl`	**`isnan(3C)`**	Provide the class to which the number belongs.
`fpgetieee`	**`fpgetieee(3C)`**	Get the current IEEE mode bit.
`fpgetmask`	**`fpgetmask(3C)`**	Get the current exceptions mask.
`fpgetround`	**`fpgetround(3C)`**	Get the current rounding mode.
`fpgetsticky`	**`fpgetsticky(3C)`**	Get the current exceptions sticky flags.
`fpsetieee`	**`fpsetieee(3C)`**	Set the current IEEE mode.
`fpsetmask`	**`fpsetmask(3C)`**	Set the current exceptions mask.
`fpsetround`	**`fpsetround(3C)`**	Set the current rounding mode.

Table 16-50. Floating-Point Environment Functions (Cont.)

Function	Reference	Brief Description
`fpsetsticky`	**`fpsetsticky(3C)`**	Set the current exceptions sticky flags.
`isnan, isnand, isnanf`	**`isnan(3C)`**	Determine if the number is a NaN.
`unordered, unorderedl`	**`isnan(3C)`**	Determine if the numbers are unordered.

Pseudo-Random Number Generation Functions

The following functions generate pseudo-random numbers.

Table 16-51. Pseudo-Random Number Generation Functions

Function	Reference	Brief Description
`drand48`	**`drand48(3C)`**	Obtain a random double-precision value over the interval (0 to 1).
`erand48`	**`drand48(3C)`**	Obtain a random double-precision value over the interval (0 to 1), but without the need for an initialization entry point.
`jrand48`	**`drand48(3C)`**	Generate a random integer value over the interval (-2**32-1 to 2**32-1), but without the need for an initialization entry point.
`lcong48`	**`drand48(3C)`**	Set the parameters for `drand48`, `lrand48`, and `mrand48`.
`initstate`	**`random(3C)`**	Initialize a state array.
`lrand48`	**`drand48(3C)`**	Generate a random integer value over the interval (0 to 2**32-1).
`mrand48`	**`drand48(3C)`**	Generate a random integer value over the interval (-2**32-1 to 2**32-1).
`nrand48`	**`drand48(3C)`**	Generate a random integer value over the interval (0 to 2**32-1), but without the need for an initialization entry point.
`rand, rand_r`	**`rand(3C)`**	Generate a random integer value over the interval (0 to 32767).
`random`	**`random(3C)`**	Generate a random integer value over the interval (0 to 2**32-1).
`seed48`	**`drand48(3C)`**	Seed the generator for `drand48`, `lrand48`, and `mrand48`.
`setstate`	**`random(3C)`**	Set a state array.

Table 16-51. Pseudo-Random Number Generation Functions (Cont.)

Function	Reference	Brief Description
`srand`	**`rand(3C)`**	Seed the generator for `rand`.
`srandom`	**`random(3C)`**	Seed the generator for `random`.
`srand48`	**`drand48(3C)`**	Seed the generator for `drand48`, `lrand48`, and `mrand48` using a long integer.

Programs

These functions provide control over a running program and access to its invocation environment. They are grouped according to the following categories:

- "Flow" on page 16-44
- "Profile" on page 16-44
- "Parameters" on page 16-45

Flow

These functions provide control over the flow of a program.

Table 16-52. Flow Functions

Function	Reference	Brief Description
`atexit`	**`atexit(3C)`**	Add a program termination routine.
`longjmp`	**`setjmp(3C)`**	Restore the environment saved by `setjmp`.
`setjmp`	**`setjmp(3C)`**	Save the environment for later use by `longjmp`.
`siglongjmp`	**`sigsetjmp(3C)`**	Restore the environment saved by `sigsetjmp`.
`sigsetjmp`	**`sigsetjmp(3C)`**	Save the environment, with signal state, for later use by `siglongjmp`.

Profile

These functions prepare an execution profile of a program.

Table 16-53. Profile Functions

Function	Reference	Brief Description
`monitor`	**`monitor(3C)`**	Cause the process to record a histogram of the program counter location.
`profil`	**`profil(2)`**	Provide an execution time profile.

Parameters

These functions support the getting and setting of program invocation arguments and environment information.

Table 16-54. Parameters Functions

Function	Reference	Brief Description
`getopt`	**`getopt(3C)`**	Get the next option letter from the option vector.
`getsubopt`	**`getsubopt(3C)`**	Parse suboptions in a flag argument initially parsed by `getopt`.
`getcwd`	**`getcwd(3C)`**	Get the path name of the current directory.
`getenv`	**`getenv(3C)`**	Obtain the string value associated with an environment variable.
`getpass, getpass_r`	**`getpass(3C)`**	Read a string from the terminal without echoing.
`getwd`	**`getwd(3C)`**	Get the path name of the current directory.
`putenv`	**`putenv(3C)`**	Change or add the value of an environment variable.

Processes

These functions provide control over the IEEE floating-point environment used by the program. They are grouped according to the following categories:

- “Control” on page 16-46
- “Signals” on page 16-47
- “User-Level Interrupts” on page 16-49
- “Lightweight Processes” on page 16-49

Control

These functions support operations on processes and control of processes.

Table 16-55. Control Functions

Function	Reference	Brief Description
`abort`	**`abort(3C)`**	Cause an IOT signal to be sent to the process.
`alarm`	**`alarm(2)`**	Set the process' alarm clock.
`cuserid`	**`cuserid(3S)`**	Indicate the login name for the owner of the current process.
`execl`, `execle`, `execlp`, `execv`, `execve`, `execvp`	**`exec(2)`**	Overlay a process image on an old process.
`exit`, `_exit`	**`exit(2)`**	Terminate a process.
`ftok`	**`stdipc(3C)`**	Create a key for use by the inter-process communication facilities.
`fork`, `fork1`, `forkall`	**`fork(2)`**	Create a new process.
`getcontext`	**`getcontext(2)`**	Get a user-level context.
`getegid`	**`getuid(2)`**	Get the effective group ID of the calling process.
`geteuid`	**`getuid(2)`**	Get the effective user ID of the calling process.
`getgid`	**`getuid(2)`**	Get the real group ID of the calling process.
`getpgid`	**`getpid(2)`**	Get the process group ID of the calling process.
`getpgrp`	**`getpid(2)`**	Get the process group ID of the calling process.
`getpid`	**`getpid(2)`**	Get the process ID of the calling process.
`getppid`	**`getpid(2)`**	Get the parent process ID of the calling process.
`getsid`	**`getsid(2)`**	Get the session ID of the calling process.
`getuid`	**`getuid(2)`**	Get the real user ID of the calling process.
`kill`	**`kill(2)`**	Send a signal to a process or group of processes.
`makecontext`	**`makecontext(3C)`**	Make a user-level context.
`nanosleep`	**`nanosleep(3C)`**	Suspend execution of current process for an interval, using high-resolution timing.
`nice`	**`nice(2)`**	Change the priority of a time-sharing process.
`pause`	**`pause(2)`**	Suspend the process until a signal is received.
`priocntl`	**`priocntl(2)`**	Control the scheduling of active processes.

Table 16-55. Control Functions (Cont.)

Function	Reference	Brief Description
priocntlset	**priocntlset(2)**	Change the scheduling properties of running processes.
processor_bind	**processor_bind(3C)**	Bind a process or LWP(s) to a specific processor.
procpriv	**procpriv(2)**	Control privileges associated with the calling process.
procprivl	**procprivl(3C)**	Control privileges associated with the calling process.
ptrace	**ptrace(2)**	Trace a process.
setcontext	**setcontext(2)**	Set a user-level context.
setgid	**setuid(2)**	Set the real group ID of the calling process.
setpgid	**setpgid(2)**	Set the process group ID of the calling process.
setpgrp	**setpgrp(2)**	Set the process group ID of the calling process.
setgid	**setsid(2)**	Set the session ID of the calling process.
setuid	**setuid(2)**	Set the real user ID of the calling process.
tcsetpgrp	**tcsetpgrp(3C)**	Set a terminal foreground process group ID.
sleep	**sleep(3C)**	Suspend execution of current process for an interval.
swapcontext	**swapcontext(3C)**	Swap a user-level context.
system	**system(3S)**	Execute a shell command.
vfork	**vfork(2)**	Spawn a new process efficiently.
wait	**wait(2)**	Wait for a child process to stop or terminate.
waitid, waitpid	**waitid(2)**	Wait for a child process to change state.

Signals

These functions support the use of signals.

Table 16-56. Signals Functions

Function	Reference	Brief Description
bsd_signal	**bsd_signal(3C)**	Alternative to **signal(2)**.
gsignal	**ssignal(3C)**	Send a software signal.
psiginfo	**psignal(3C)**	Write a signal message to standard error.
psignal	**psignal(3C)**	Write a signal message to standard error.
sig2str	**st2sig(3C)**	Obtain the suffix name of a system signal.
sigaction	**sigaction(2)**	Perform detailed signal management.
sigaddset	**sigsetops(3C)**	Add a signal to a set.

Table 16-56. Signals Functions (Cont.)

Function	Reference	Brief Description
`sigalstack`	**`sigalstack(2)`**	Get or set a signal alternate stack context.
`sigdelset`	**`sigsetops(3C)`**	Delete a signal from a set.
`sigemptyset`	**`sigsetops(3C)`**	Exclude from a set all signals defined by the system.
`sigfillset`	**`sigsetops(3C)`**	Include in a set all signals defined by the system.
`sighold`	**`signal(2)`**	Add a signal to the calling process' signal mask.
`sigignore`	**`signal(2)`**	Set the disposition of a signal to SIG_IGN.
`sigismember`	**`sigsetops(3C)`**	Determine if a signal is in a set.
`signal`	**`signal(2)`**	Modify signal disposition.
`sigpause`	**`signal(2)`**	Remove a signal from the calling process' signal mask, and suspend the calling process.
`sigpending`	**`sigpending(2)`**	Obtain signals that are blocked and pending.
`sigprocmask`	**`sigprocmask(2)`**	Examine and/or change the calling process' signal mask.
`sigrelse`	**`signal(2)`**	Remove a signal from the calling process' signal mask.
`sigsend, sigsendset`	**`sigsend(2)`**	Send a signal to a process or group of processes.
`sigset`	**`signal(2)`**	Add a signal to the calling process' signal mask before executing the signal handler.
`sigsuspend`	**`sigsuspend(2)`**	Install a signal mask and suspend the calling process.
`ssignal`	**`ssignal(3C)`**	Arrange for handling of software signals.
`sigsendset`	**`sigsend(2)`**	Provides an alternate interface for sending signals to sets of processes.
`sigsend`	**`sigsend(2)`**	Send a signal to a process or group of processes.
`sigwait`	**`sigwait(2)`**	Wait for a signal to be posted.
`ssignal`	**`ssignal(3C)`**	Arrange for handling of software signals.
`str2sig`	**`st2sig(3C)`**	Obtain the number of a system signal.

User-Level Interrupts

These functions support the use of user-level interrupts.

Table 16-57. User-Level Interrupts Functions

Function	Reference	Brief Description
`iconnect`	**`iconnect(3C)`**	Provide a user-level interrupt connection.
`ienable`	**`ienable(3C)`**	Enable a user-level interrupt.

Lightweight Processes

These functions support lightweight processes (LWPs).

Table 16-58. Lightweight Processes Functions

Function	Reference	Brief Description
`_lwp_cond_broadcast`	**`_lwp_cond_broadcast(2)`**	Wake up all LWPs waiting on a condition.
`_lwp_cond_signal`	**`_lwp_cond_signal(2)`**	Wake up a single LWP waiting on a condition.
`_lwp_cond_timedwait`	**`_lwp_cond_timedwait(2)`**	Wait on a condition variable for a limited time.
`_lwp_cond_wait`	**`_lwp_cond_wait(2)`**	Wait on a condition.
`_lwp_continue`	**`_lwp_continue(2)`**	Continue LWP execution.
`_lwp_create`	**`_lwp_create(2)`**	Create a new LWP.
`_lwp_exit`	**`_lwp_exit(2)`**	Terminate the calling LWP.
`_lwp_getprivate`	**`_lwp_getprivate(2)`**	Get an LWP-specific reference.
`_lwp_global_self`	**`_lwp_global_self(2)`**	Get the current LWP's global identifier.
`_lwp_info`	**`_lwp_info(2)`**	Get time-accounting information of a single LWP.
`_lwp_kill`	**`_lwp_kill(2)`**	Send a signal to a sibling LWP.
`_lwp_makecontext`	**`_lwp_makecontext(2)`**	Make an LWP context.
`_lwp_mutex_lock`	**`_lwp_mutex_lock(2)`**	Lock a mutex on behalf of the calling LWP.
`_lwp_mutex_trylock`	**`_lwp_mutex_trylock(2)`**	Conditionally lock a mutex on behalf of the calling LWP.
`_lwp_mutex_unlock`	**`_lwp_mutex_unlock(2)`**	Unlock a mutex.
`_lwp_self`	**`_lwp_self(2)`**	Provide the current LWP's identifier.
`_lwp_sema_init`	**`_lwp_sema_init(2)`**	Initialize a semaphore.
`_lwp_sema_post`	**`_lwp_sema_post(2)`**	Release a semaphore.
`_lwp_sema_trywait`	**`_lwp_sema_trywait(2)`**	Conditionally acquire a semaphore.

Table 16-58. Lightweight Processes Functions (Cont.)

Function	Reference	Brief Description
_lwp_sema_wait	**_lwp_sema_wait(2)**	Acquire a semaphore.
_lwp_setprivate	**_lwp_setprivate(2)**	Set an LWP-specific reference.
_lwp_suspend	**_lwp_suspend(2)**	Suspend LWP execution.
_lwp_wait	**_lwp_wait(2)**	Wait for termination of a sibling LWP.
client_block	**client_block(2)**	Block a client LWP and establish a server LWP.
client_wake1	**client_block(2)**	Wake a client LWP.
client_wakechan	**client_block(2)**	Wake all client LWPs on a chain.
cpu_bias	**cpu_bias(2)**	Control CPU biasing and assignment for LWPs.
priocntllist	**priocntllist(2)**	Control the scheduling of active processes for a set of LWPs.
server_block	**server_block(2)**	Block a server LWP.
server_wake1	**server_block(2)**	Wake a blocked server LWP.
server_wakechan	**server_block(2)**	Wake all blocked server LWPs on a chain.

Security

These functions support user- and system-level security. They are grouped into the following categories:

- “Access Control Lists” on page 16-51
- “Auditing” on page 16-51
- “Levels” on page 16-51
- “Other Security” on page 16-52
- “Encryption and Decryption” on page 16-52

Access Control Lists

These functions access Access Control Lists (ACLs).

Table 16-59. Access Control Lists Functions

Function	Reference	Brief Description
`acl`	**`acl(2)`**	Set a file's ACL.
`aclipc`	**`aclipc(2)`**	Get or set an IPC object's ACL.
`aclsort`	**`aclsort(3C)`**	Sort an ACL.

Auditing

These functions support auditing operations.

Table 16-60. Auditing Functions

Function	Reference	Brief Description
`auditbuf`	**`auditbuf(2)`**	Get or set audit buffer attributes.
`auditctl`	**`auditctl(2)`**	Get or set the status of auditing.
`auditdmp`	**`auditdmp(2)`**	Write an audit record to an audit buffer.
`auditevt`	**`auditevt(2)`**	Get or set auditable events.
`auditlog`	**`auditlog(2)`**	Get or set audit log file attributes.

Levels

These functions control levels.

Table 16-61. Levels Functions

Function	Reference	Brief Description
`lvldom`	**`lvldom(2)`**	Determine the domination relationship of two levels.
`lvlequal`	**`lvlequal(2)`**	Determine the equality of two levels.
`lvlfile`	**`lvlfile(2)`**	Get or set the level of a file.
`lvlin`	**`lvlin(3C)`**	Translate a level from text format to internal format.
`lvlintersect`	**`lvlintersect(3C)`**	Perform the intersection of two levels.
`lvlipc`	**`lvlipc(2)`**	Manipulate an IPC object's level.

Table 16-61. Levels Functions (Cont.)

Function	Reference	Brief Description
`lvlout`	**`lvlout(3C)`**	Translate a level from internal format to text format.
`lvlproc`	**`lvlproc(2)`**	Get or set the level of a process.
`lvlunion`	**`lvlunion(3C)`**	Perform the union of two levels.
`lvlvalid`	**`lvlvalid(3C)`**	Check the validity of a level.
`lvlvfs`	**`lvlvfs(2)`**	Get or set the level ceiling of a mounted file system.

Other Security

These functions support miscellaneous security operations.

Table 16-62. Other Security Functions

Function	Reference	Brief Description
`initgroups`	**`initgroups(3C)`**	Initialize the supplementary group access list.
`mkmld`	**`mkmld(2)`**	Make a Multilevel Directory.
`mldmode`	**`mldmode(2)`**	Get or set the Multilevel Directory mode of a process.
`secadvise`	**`secadvise(2)`**	Obtain kernel advisory access information.
`secsys`	**`secsys(2)`**	Initialize enhanced security.

Encryption and Decryption

The following functions allow access to the Data Encryption Standard (DES) algorithm and other encryption/decryption algorithms.

Table 16-63. Encryption and Decryption Functions

Function	Reference	Brief Description
`crypt`	**`crypt(3C)`**	Encode a string.
`encrypt`	**`crypt(3C)`**	Encode/decode a string.
`isencrypt`	**`isencrypt(3G)`**	Determine if a character buffer is encrypted.
`setkey`	**`crypt(3C)`**	Initialize a key for subsequent use by `encrypt`.

System Environment

These functions provide support operations that access and control system-wide resources and configurations. They are grouped into the following categories:

- "Loadable Kernel Modules" on page 16-53
- "Other System Environment" on page 16-53

Loadable Kernel Modules

These functions provide control over loadable kernel modules.

Table 16-64. Loadable Kernel Modules Functions

Function	Reference	Brief Description
`modload`	**`modload(2)`**	Load a loadable kernel module on demand.
`modpath`	**`modpath(2)`**	Change the search path for loadable kernel modules.
`modstat`	**`modstat(2)`**	Get information for loadable kernel modules.
`moduload`	**`moduload(2)`**	Unload a loadable kernel module on demand.

Other System Environment

These functions support other operations on the system-wide environment.

Table 16-65. Other System Environment Functions

Function	Reference	Brief Description
`access`	**`access(2)`**	Enable or disable process accounting.
`eti_map`	**`eti_request(3C)`**	Map an edge-triggered interrupt into the process' address space.
`eti_request`	**`eti_request(3C)`**	Issue a control operation to an edge-triggered interrupt.
`eti_unmap`	**`eti_unmap(3C)`**	Detach a shared memory region from a process.
`getpagesize`	**`getpagesize(3C)`**	Get the system page size.
`getgroups`	**`getgroups(2)`**	Get supplementary group access list IDs.
`getksym`	**`getksym(2)`**	Get information for a global kernel symbol.
`getrlimit`	**`getrlimit(2)`**	Get a maximum system resource consumption limit.
`hrdclk`	**`hrdclk(2)`**	Control hardclock interrupt handling.
`keyctl`	**`keyctl(2)`**	Get and set user and processor limits.
`mpadvise`	**`mpadvise(3C)`**	Provide multiprocessor control.

Table 16-65. Other System Environment Functions (Cont.)

Function	Reference	Brief Description
processor_info	**processor_info(2)**	Provide information about a processor.
resched_cntl	**resched_cntl(2)**	Provide CPU rescheduling control.
setgroups	**getgroups(2)**	Set supplementary group access list IDs.
setrlimit	**setrlimit(2)**	Set a maximum system resource consumption limit.
swapctl	**swapctl(2)**	Manage swap space.
sysconf	**sysconf(3C)**	Provide the value of a configurable system variable.
syscx	**syscx(2)**	Perform machine-specific functions.
sync	**sync(2)**	Update a super block.
sysinfo	**sysinfo(2)**	Get and set system information strings.
uadmin	**uadmin(2)**	Control basic administrative operations.
ulimit	**ulimit(2)**	Get and set user limits.
umask	**umask(2)**	Get and set the file creation mask.
uname	**uname(2)**	Obtain the name of the current UNIX system.
vme_address	**vme_address(3C)**	Obtain a (H)VME physical address.

17
Floating-Point Operations

17
Floating-Point Operations

Introduction

The supporting hardware platforms support the *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Standard 754-1985). Concurrent Computer Corporation's compilation systems use the IEEE standard single- and double-precision data types, operations, and conversions specified in this standard. Library functions are provided for further IEEE support.

You will probably not need any special functions to use floating-point operations in your programs. If you do, however, you can find information about floating-point support in this chapter. (For more details on how the compilation systems support the IEEE standard see "IEEE Requirements" on page 17-11.)

This chapter contains sections on the following topics:

- The details of IEEE arithmetic
- Floating-point exception handling
- Single-precision floating-point operations
- Implicit precision of subexpressions
- IEEE requirements

If your code depends on a side effect of a floating-point operation (such as the setting of a trap), note that the optimizer may remove the floating-point operation if the result of the operation is not used elsewhere. Therefore, your process may never see the side effect it depends on. For example, if your program depends on a trap resulting from the following operation:

```
x = a + b
```

and the operation is removed by the optimizer because the result is not used anywhere else, the trap never occurs.

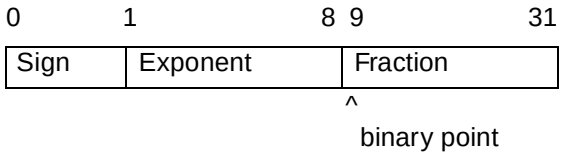
IEEE Arithmetic

This section provides the details of floating-point representation and exception handling. Most users need not be concerned with the details of the floating-point environment. Floating-point formats, values, and operations are based on the *IEEE Standard for Binary Floating-Point Arithmetic,* ANSI/IEEE Standard 754-1985.

Data Types and Formats

Single-Precision

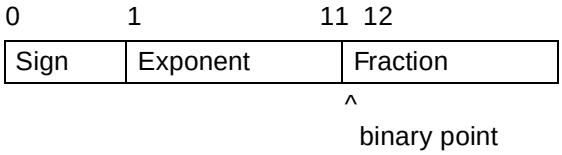
Single-precision floating-point numbers have the following format:



Field	Bit Position	Full Name
Sign	0	Sign bit (0==positive, 1==negative)
Exponent	1-8	Exponent (biased by 127)
Fraction	9-31	Fraction (bits to right of binary point)

Double-Precision

Double-precision floating-point numbers have the following format:



Field	Bit Position	Full Name
Sign	0	Sign bit (0==positive, 1==negative)
Exponent	1-11	Exponent (biased by 1023)
Fraction	12-63	Fraction (bits to right of binary point)

Language Mappings

The IEEE single- and double-precision data types are denoted by the following language data types.

Data Type	C	Fortran	Ada
Single	`float`	`REAL`	`float (digits 1..9)`
		`REAL*4`	
Double	`double`	`DOUBLE PRECISION`	`long_float (digits 10..16)`
		`REAL*8`	

Normalized Numbers

A number is normalized if the exponent field contains other than all 1's or all 0's. The exponent field contains a biased exponent, where the bias is 127 in single-precision, and 1023 in double-precision. Thus, the exponent of a normalized floating-point number is in the range -126 to 127, inclusive, for single-precision, and in the range -1022 to 1023, inclusive, for double-precision.

There is an implicit bit associated with both single- and double-precision formats. The implicit bit is not explicitly stored anywhere (thus its name). Logically, for normalized operands the implicit bit has a value of 1 and resides immediately to the left of the binary point (in the 2^0 position). Thus the implicit bit and fraction field together can represent values in the range 1 to $2 - 2^{-23}$, inclusive, for single-precision, and in the range 1 to $2 - 2^{-52}$, inclusive, for double-precision.

Thus normalized single-precision numbers can be in the range (plus or minus) 2^{-126} to $(2 - 2^{-23}) \times 2^{127}$, inclusive.

Normalized double-precision numbers can be in the range (plus or minus) 2^{-1022} to $(2 - 2^{-52}) \times 2^{1023}$, inclusive.

Denormalized Numbers

A number is denormalized if the exponent field contains all 0's and the fraction field does not contain all 0's.

Thus denormalized single-precision numbers can be in the range (plus or minus) $2^{-126} \times 2^{-22} = 2^{-148}$ to $(1 - 2^{-22}) \times 2^{-126}$, inclusive.

Denormalized double-precision numbers can be in the range (plus or minus) $2^{-1022} \times 2^{-51} = 2^{-1073}$ to $(1 - 2^{-51}) \times 2^{-1022}$, inclusive.

Both positive and negative zero values exist, but they are treated the same during floating-point calculations.

Maximum and Minimum Representable Floating-Point Values

The maximum and minimum representable values in floating-point format are defined in the C header file `values.h`. They evaluate to the following values:

Symbolic Constant	Value
`MAXDOUBLE`	`1.79769313486231470e+308`
`MAXFLOAT`	`((float)3.40282346638528540e+38)`
`MINDOUBLE`	`2.22507385850720270e-308`
`MINFLOAT`	`((float)1.17549435082228740e-38)`

The Fortran run-time library provides functions which return these values. Refer to **`flmin(3F)`** for further information.

Refer to Appendix F in the *HAPSE Reference Manual* for the use and values of the model numbers of floating-point type.

Special-Case Values

The following table gives the names of special cases and how each is represented.

Value Name	Sign	Exponent	Fraction	
			MSB	Rest of Fraction
NaN (non-trapping)	X	Max	0	Nonzero
Trapping NaN	X	Max	1	X
Positive Infinity	0	Max	Min	
Negative Infinity	1	Max	Min	
Positive Zero	0	Min	Min	
Negative Zero	1	Min	Min	
Denormalized Number	X	Min	Nonzero	
Normalized Number	X	NotMM	X	

Key:

X	Does not matter
Max	Maximum value that can be stored in the field (all 1’s)
Min	Minimum value that can be stored in the field (all 0’s)
NaN	Not a number

NotMM Field is not equal to either Min or Max values

Nonzero Field contains at least one “1” bit

MSB Most Significant Bit

The algorithm for classification of a value into special cases follows:

```
If (Exponent==Max)
    If (Fraction==Min)
        Then the number is Infinity (Positive or Negative
        as determined by the Sign bit).
    Else the number is NaN (Trapping if FractionMSB==0,
    non-Trapping if FractionMSB==1).


Else If (Exponent==Min)
    If (Fraction==Min)
        Then the number is Zero (Positive or Negative
        as determined by the Sign bit).
    Else the number is Denormalized.
Else the number is Normalized.
```

NaNs and Infinities

The floating-point system supports two special representations:

- *Infinity* - Positive infinity in a format compares greater than all other representable numbers in the same format. Arithmetic operations on infinities are quite intuitive. For example, adding any representable number to infinity is a valid operation, the result of which is positive infinity. Subtracting positive infinity from itself is invalid. If some arithmetic operation overflows, and the overflow trap is disabled, in some rounding modes the result is infinity.
- *Not-a-Number* (*NaN*) - These floating-point representations are not numbers. They can be used to carry diagnostic information. There are two kinds of NaNs: signaling NaNs and quiet NaNs. Signaling NaNs raise the invalid operation exception whenever they are used as operands in floating-point operations. Quiet NaNs propagate through most operations without raising any exception. The result of these operations is the same quiet NaN. NaNs are sometimes produced by the arithmetic operations themselves. For example, 0.0 divided by 0.0, when the invalid operation trap is disabled, produces a quiet NaN.

The C header file **`ieeefp.h`** defines the interface for the floating-point exception and environment control. This header defines three interfaces:

- Rounding Control
- Exception Control
- Exception Handling

The Fortran compilation system provides intrinsic functions for compile-time generation of NaNs for `REAL` and `COMPLEX` data types. Refer to **`nan(3F)`** and **`hf77(1)`** for more information.

Rounding Control

The floating-point arithmetic provides four rounding modes that affect the result of most floating-point operations. (These modes are defined in the header **`ieeefp.h`**):

`FP_RN`	Round to nearest representable number, tie -> even
`FP_RP`	Round toward plus infinity
`FP_RM`	Round toward minus infinity
`FP_RZ`	Round toward zero (truncate)

You can check the current rounding mode with the function

```
fp_rnd fpgetround(void); /*return current rounding mode*/
```

You can change the rounding mode for floating-point operations with the function:

```
fp_rnd fpsetround(fp_rnd);
            /* set rounding mode, return previous */
```

(`fp_rnd` is an enumeration type with the enumeration constants listed and described above. The values for these constants are in **`ieeefp.h`**.) Alternatively, this can be done with the **`-Qfpcr`** linker option; see “Using the Link Editor” on page 4-1 for details.

The examples in this section, such as the one directly above, illustrate function prototypes. For information on function prototypes, see the Concurrent *C Reference Manual*.

The default rounding mode is round-to-nearest. In C and Fortran, floating-point to integer conversions are always done by truncation, and the current rounding mode has no effect on these operations.

(For more information, see the **`fpgetround(3C)`** and **`fpsetround(3C)`** manual pages.)

Floating-Point Exceptions

Floating-point exception interrupts are enabled, and they operate in imprecise mode by default on the supporting hardware platforms for C and Fortran programs. Ada programs generate the exceptions if checks are not suppressed. The supporting hardware platforms

provide the ability to enable or disable floating-point exceptions, as well as to specify whether the exceptions are precise or imprecise. If this interrupt is enabled, the operating system will receive a `SIGFPE` signal any time an enabled floating-point exception is raised by the hardware. A floating-point exception is enabled if its corresponding bit is on in the `fpcsr` register. If this interrupt is disabled, the operating system will not be notified when a floating-point exception is raised by the hardware. If an exception is imprecise, it may not be possible for a program to recover from the exception because the system provides insufficient information for doing so. Complete information is provided for a precise exception.

The disabling of this interrupt provides for improved performance. Use of imprecise exceptions rather than precise exceptions provides for improved performance when floating-point exceptions are enabled. By default, programs run in an `imprecise exceptions` mode. Concurrent Computer Corporation's compilation systems provide two ways of creating programs which will execute with floating-point exceptions disabled or enabled as precise or imprecise. One way is to use the **`-Qflttrap`** option with the C compiler, **`cc(1)`**, or the Fortran compiler, **`f77(1)`**. This option directs the compilers to produce additional code to detect and trap floating-point exceptions. The other way is to use the **`-Qfpexcept=`** option with the link editor, **`ld(1)`**. This option directs the link editor to set the `fp_except_kind` field in the program's vendor section. The kernel sets bits 52 and 55 of the `msr` register, at program start up, based upon the setting of the `fp_except_kind` field.

Exceptions, Status Bits, and Control Bits

Floating-point operations can lead to any of the following types of floating-point exceptions:

Divide by zero	This exception happens when a non-zero number is divided by floating-point zero.
Invalid operation	All operations on signaling NaNs raise an invalid operation exception. Zero divided by zero, infinity subtracted from infinity, and infinity divided by infinity all raise this exception. When a quiet NaN is compared with the greater or lesser relational operators, an invalid operation exception is raised.
Overflow	This exception occurs when the result of any floating-point operation is too large in magnitude to fit in the intended destination.
Underflow	When the underflow trap is enabled, an underflow exception is signaled when the result of some operation is a very tiny non-zero number that may cause some other exception later (such as overflow upon division). When the underflow trap is disabled, an underflow exception occurs only when both the result is very tiny (as explained above) and a loss of accuracy is detected.
Inexact or imprecise	This exception is signaled if the rounded result of an operation is not identical to the infinitely precise result. Inexact exceptions are quite common. 1.0 / 3.0 is an inexact operation. Inexact exceptions also occur when the operation overflows without an overflow trap. The above examples for the exception types do not con-

stitute an exhaustive list of the conditions when an exception can occur.

Whenever an exception occurs, a corresponding status bit is set (=1) for that exception. On the supporting hardware platforms, these bits are contained in the `fpscr` register. When status bits are set by the hardware and/or operating system, they remain set until cleared by user software.

You can check the status of the status bits by using the function

```
fp_except fpgetsticky(void);
                /* return logged exceptions */
```

`fp_except` is an enumeration type that can have any combination of the following constant values:

`FP_X_DZ`	Divide-by-zero exception
`FP_X_INV`	Invalid operation exception
`FP_X_OFL`	Overflow exception
`FP_X_UFL`	Underflow exception
`FP_X_IMP`	Imprecise (loss of precision)

(The values for these constants are in **`ieeefp.h`**.)

You can change the status bits by using the function

```
fp_except fpsetsticky(fp_except);
    /* set logged exceptions, return previous */
```

There is also a control bit (mask bit) associated with each exception. On the supporting hardware platforms, these bits are contained in the `fpscr` register. When an exception occurs, if the corresponding control bit is enabled (=1), a trap occurs. When a trap occurs, the result of the operation is not written and a signal is sent to the user process. You can check the status of these mask bits by using the function

```
fp_except fpgetmask(void);   /* current exception mask */
```

You can also selectively enable or disable any of the exceptions by calling the function

```
fp_except fpsetmask(fp_except);
                    /* set mask, return previous */
```

with appropriate mask values.

In Ada programs, a `numeric_error` is raised for each of these exceptions except `underflow`.

By default, programs built with Concurrent Computer Corporation's compilation systems will begin execution having only the underflow and imprecise exception control bits masked off.

Alternatively, this can be done with the **`-Qfpcr`** linker option; see "Using the Link Editor" on page 4-1 for details. For more information, see the following manual pages:

fpgetsticky(3C),
fpsetsticky(3C),
fpgetmask(3C),
fpsetmask(3C),
and
fpgetround(3C)

Exception Handling

If a floating-point trap is enabled, your process is signaled when the corresponding floating-point exception occurs. PowerUX signals your process by sending `SIGFPE`. If you intend to handle the exception, you must specify a handler for `SIGFPE`. You can specify the handler by calling the C `signal()` routine as follows:

```
#include <signal.h>

extern void myhandler ();

foo ()
{
    (void) signal (SIGFPE, myhandler);
}
```

The Fortran compilation system also provides a `signal` function. Refer to **signal(3F)** for more information.

Ada users who set up a signal handler should note that the Ada executive reserves `SIGFPE`. Use of a signal handler for `SIGFPE` will cause non-standard behavior in Ada programs.

Single-Precision Floating-Point Operations

The ANSI standard for C has a provision that allows expressions to be evaluated in single-precision arithmetic if there is no `double` (or `long double`) operand in the expression. The C compiler supports this provision.

Floating-point constants are double-precision, unless explicitly stated to be `float`. For example, in the statements

```
float a,b;
 ...
a = b + 1.0;
```

because the constant `1.0` has type `double`, `b` is promoted to `double` before the addition and the result is converted back to `float`. However, the constant can be made explicitly a `float`:

```
a = b + 1.0f;
```

or

```
a = b + (float) 1.0;
```

In this case, the statement can potentially be compiled to a single instruction. Single-precision operations tend to be faster than double-precision operations.

Whether a computation can be done in single-precision is decided based on the operands of each operator. Consider the following:

```
float s;
double d;
d = d + s * s;
```

`s * s` is computed to produce a single-precision result, which is promoted to double-precision and added to `d`.

The IEEE P854 task force responsible for format independent floating-point environment issues may disallow the multiplication to be carried in single-precision in this context.

Note that using single-precision (as versus double-precision) arithmetic can result in loss of precision, as illustrated in the following example.

```
float f  = 8191.f * 8191.f;    /* evaluate as a float  */
double d = 8191.  * 8191. ;    /* evaluate as a double */
printf ("As float:  %f\nAs double: %f\n", f, d);
```

The result is:

```
As float:  67092480.000000
As double: 67092481.000000
```

Also, `long int` variables (same as `int`) have more precision than `float` variables. Consider the following example:

```
int i,j;
i = 0x7ffffff;
j = i * 1.0;
printf("j = %x\n", j);
j = i * 1.0f;
printf("j = %x\n", j);
```

The first `printf()` statement outputs `7ffffff`, while the second prints `0`. The second `printf()` prints `0` because the nearest float to `0x7fffffff` has a value of `0x80000000`. When the value is converted to an integer, the result is `0`, and a floating-point imprecise result exception occurs. A trap occurs if this exception was enabled.

A function that is declared to return a `float` may actually return either a `float` or a `double`. If the function declaration is a prototype declaration in which at least one of the parameters is `float`, the function returns a `float`. Otherwise, it returns a `double` with precision limited to that of a `float`. (All of this is transparent.) For example:

```
float retflt(float);    /* actually returns a float  */
float retdbl1();        /* actually returns a double */
float retdbl2(int);     /* actually returns a double */
```

Arguments work as follows:

```
double takeflt(float x);    /* takes a float  */

double takedbl(x)
float x;                    /* takes a double */
```

Single-Precision Functions

The system math libraries (**`libm.a`** and **`libM.a`**) contain single-precision versions of several functions. These floating-point functions all have names that end in `f`, take and return `floats`, and do most internal computations in single-precision arithmetic. For a complete list of floating-point functions in the math libraries, see Chapter 16 (“Run-Time Libraries”).

The Ada package `math` includes **`libm.a`**.

Double-Extended-Precision

Concurrent Computer Corporation’s compilation systems do not produce code that uses IEEE double-extended-precision arithmetic, either for intermediate or final results. All results are computed with the precision implicit in their type.

The C `long double` data type is computationally equivalent to the `double` data type. In the future, `long double` may be used for double-extended-precision values; therefore, it is best to avoid using `long double`, for compatibility reasons.

IEEE Requirements

All arithmetic computations generated by Concurrent Computer Corporation’s compilation systems strictly conform to IEEE requirements. The following is a discussion of some topics where the compilation systems fall short of completely meeting the ANSI/IEEE Standard 754-1985 requirements or the spirit of the requirements.

Conversion of Floating-Point Formats to Integer

IEEE requires floating-point to integer format conversions to be affected by the current rounding mode. However, the C and Fortran languages require these conversions to be done by truncation (which is the same as round-to-zero). In the compilation systems, floating-point to integer conversions are done by truncation.

Square Root

IEEE requires the square root of a negative non-zero number to raise invalid operation, whereas PowerUX system compatibility requires square root to return 0.0 with `errno` set to `EDOM`. The PowerUX math libraries provide this level of compatibility.

Compares and Unordered Condition

In addition to the usual relationships between floating-point values (less than, equal, greater than), there is a fourth relationship: unordered. The unordered case arises when at least one operand is a NaN. Every NaN compares unordered with any value, including itself. The C compilation system provides the following predicates required by IEEE between floating-point operands:

==	>=
!=	<
>	<=

While there is no predicate to test for unordered, you can use `isnand()` or `isnanf()` to test whether an argument is a NaN. For information on `isnand()` and `isnanf()`, see the **`isnan(3C)`** manual page.

The relations `>`, `>=`, `<`, and `<=` raise invalid operation for unordered operands. The compiler generated code does not guard against the unordered outcome of a comparison. If the trap is masked, the path taken for unordered conditions is the same as if the conditional were true, which may result in incorrect behavior.

For the predicates `==` and `!=`, unordered condition does not lead to invalid operation. The path taken for unordered condition is the same as when the operands are non-equal, which is correct.

`(a > b)` is not the same as `( !(a <= b) )` in IEEE floating-point arithmetic. The difference occurs when `b` or `a` compares unordered. The C compiler generates the same code for both cases.

NaNs and Infinities in Input/Output

The `printf()` family of functions prints NaNs or infinities as symbolic names. Ideally, whatever `printf()` outputs, `scanf()` should be able to read using the same format. However, `scanf()` does not recognize NaNs and infinities for floating-point formats. Since these special cases serve mostly as diagnostics for erroneous floating-point computation, outputting these cases was considered more important than being able to read them.

18
Inter-Language Interfacing

18
Inter-Language Interfacing

Introduction

Calling subroutines written in one language from routines written in another requires a knowledge of calling conventions and data types specific to the architecture on which the program will run and the languages the program is written in. This chapter discusses inter-language interfacing between C and Fortran on the supporting hardware platforms. For more information about C, see the *Concurrent C Reference Manual*. For more information about Fortran, see the *hf77 Fortran Reference Manual.*

For information about inter-language interfacing with Ada, see the *HAPSE Reference Manual*.

Subroutine Linkage

The Stack Frame

Every routine's stack frame has the following three areas:

link area — This area occupies the lowest addresses of the stack frame and is 24 bytes in size. It holds the return address sometimes. Other words in it are reserved for future use.

parameter area — This area is reserved for parameters. Every parameter, even one passed in a register, is assigned space in this area. This area starts 24 bytes above the address where the stack pointer (`r1`) points and is always at least 32 bytes in size.

temp area — This area holds local variables, compiler temporaries, saved registers, etc.

Table 18-1 illustrates stack frame layout.

Table 18-1. Stack Frame

	Size in Bytes	Contents
High Address	32+	Caller's parameter area
	24	Caller's link area
	Any	Callee's temp area
	32+	Callee's parameter area
Low Address	24	Callee's link area

If a routine needs no temp area and does not call another subroutine, it is acceptable to have a zero-sized stack frame.

Parameters

The first thirteen floating-point parameters are passed in floating-point registers `f1` through `f13`. Integer, character, pointer and structure parameters are passed in general-purpose registers `r3` through `r10`. If there are no more parameter registers left (or, in the case of structures, not enough parameter registers left), then the parameter is passed in the parameter area.

Even when passed in a register, space exists for each parameter in the parameter area. If the parameter has alignment constraints, space in the parameter area is skipped so that the slot for the parameter in the parameter area has the appropriate alignment. This space is frequently referred to as a *hole*.

Take the following C function definition as an example:

```
f(int i1,struct {int i[10];} s1, struct {int i[2];} s2,
  double d1, float f1, int i2) {...}
```

The following table shows where each parameter gets passed and where its slot in the parameter area is:

Table 18-2. Where Parameters Are Passed

Parameter	Where Passed	Parameter Area Slot (offset in bytes)
`i1`	`r3`	0-3
`s1`	parameter area (because of general-register shortage)	4-43
`s2`	`r4`,`r5`	44-51

Table 18-2. Where Parameters Are Passed (Cont.)

Parameter	Where Passed	Parameter Area Slot (offset in bytes)
`d1`	`f1`	56-63
`f1`	`f2`	64-67
`i2`	`r6`	68-71

The length of a Fortran `CHARACTER` parameter is in a hidden extra integer parameter appended to the parameter list.

Return Values

Integer, character, and pointer values are returned in the general register `r3`. Floating-point values are returned in the floating-point register `f1`.

C `struct` and `union` return values require the caller to provide a block of memory to hold the return value. The C compiler passes the address of that block as a hidden first parameter (i.e., in general register `r3`) to the callee. When this is the case, actual parameters are passed beginning at `r4`.

Fortran `COMPLEX` return values are treated as a C `struct` consisting of two `float`s or two `double`s. Fortran `CHARACTER` return values are similar, except that the caller passes two hidden parameters to the callee: the address of the block and the size of the block.

Prologue and Epilogue

The caller places the parameters in registers or its own parameter area and executes a `bl` instruction to branch and link to the callee. The callee's prologue code then performs the following operations:

- Adjust the stack pointer (`r1`) downward to allocate space for its own stack frame. The stack pointer always maintains 16-byte alignment.
- Save the return address at offset 8 bytes in the caller's link area if the callee needs to use the `link` register.
- Save in the temp area any register that the callee is not allowed to kill but wants to use.

Before returning, each of these operations is undone in reverse order. For more information about the prologue and epilogue, see "Introduction" on page 23-1.

Register Usage

The following tables document the usage and reserved status of the various classes of registers:

Table 18-3. General Registers

Register	Use
`r0`	Scratch register (Warning: some instructions treat this register as a constant zero)
`r1`	Stack pointer
`r2`	Frame pointer, if needed for `alloca` or stack frames larger than 32K
`r3`	`int`, `char` and pointer return values; first word of non-float parameters; scratch register
`r4-r10`	Second through eighth words of non-`float` parameters; scratch registers
`r11`	Static link; scratch register
`r12-r15`	Scratch registers
`r16-r27`	Preserved registers (These registers must be saved and restored by any function that uses them.)
`r28-r30`	Reserved for post-linker optimizations
`r31`	Reserved for post-linker optimizations; thread register

Table 18-4. Floating-point Registers

Register	Use
`f0`	Scratch register
`f1`	Floating-point return value; first floating-point parameter; scratch register
`f2-f13`	Second through thirteenth floating-point parameter scratch registers
`f14-f21`	Scratch registers
`f22-f31`	Preserved registers (These registers must be saved and restored by any function that uses them)

Table 18-5. Special Registers

Register	Use
`Link`	Return address; the caller is responsible for saving and restoring this register
`Count`	Scratch register
`crf0-crf7`	All condition-register fields are scratch registers
`MQ`	Scratch register (PowerPC 601 system only)

External Names

For C, all external names appear in the object file exactly as they are spelled in the source file.

For Fortran, all external names are folded to lower case. Subroutines get a single underscore appended and common block names get two underscores appended to their names. Blank common is spelled `__BLNK__`.

Data Types

Scalar Types

The following tables give a brief description of the size and alignment constraints given various data types by default:.

Table 18-6. C Scalar Types

Type	Size	Alignment	Description
`char`	1	1	character
`short int`	2	2	integer
`long int`	4	4	integer
`float`	4	4	float
`double`	8	8	float
type `*`	4	4	pointer

On the supporting hardware platforms, the `char` type is `unsigned` by default. The **`-Qchars_signed`** option makes the default be `signed`.

The **`-W1,-7`** option to the C compiler causes `double` to be aligned to 4 bytes. This results in a minor performance penalty on the supporting hardware platforms.

Table 18-7. Fortran Scalar Types

Type	Size	Alignment	Description
`LOGICAL*1`	1	1	boolean
`LOGICAL*2`	2	2	boolean
`LOGICAL`	4	4	boolean
`INTEGER*1`	1	1	integer
`INTEGER*2`	2	2	integer
`INTEGER`	4	4	integer
`REAL`	4	4	float
`DOUBLE PRECISION`	8	8	float
`COMPLEX`	8	4	complex
`COMPLEX*16`	16	8	complex

The **`-Qalign_double=4`** option to the Fortran compiler causes `DOUBLE PRECISION` to be aligned to 4-byte boundaries. This results in a minor performance penalty on the supporting hardware platforms.

Structures

The alignment of a structure is the alignment of its most restrictive field. Padding is freely added to make each individual field maintain its alignment requirements. The size of a structure is an integer multiple of its alignment requirement. Bit fields that are of type `char`, `short`, or `long` must not cross 1, 2 or 4-byte alignment boundaries, respectively.

Common Blocks

Common blocks align their variables much like C structures. A `CHARACTER` type takes up space as though it was a `char` array. The Fortran standard requires that `DOUBLE PRECISION` be aligned to only a 4-byte boundary. Use the **`-Qalign_double=4`** option to achieve this. Beware of the minor performance penalty for doing so on the supporting hardware platforms.

5
Program Optimization

Replace with Part 5 tab

Part 5 - Program Optimization

Part 5 Program Optimization

19
Introduction to Program Optimization

19
Introduction to Program Optimization

Introduction

If you want to reduce the time your program takes to run or the resources your program uses, you should understand program optimization. This part of the manual deals with performance tuning through program optimization.

Chapter 20 (“Program Optimization”) discusses optimization concepts, options, parameters, considerations, and strategies.

For information about program performance and profiling with the **`analyze`** and **`report`** tools, see Chapter 11 (“Performance Analysis”).

20
Program Optimization

20
Program Optimization

This chapter provides an overview of the features of Concurrent's compiler technology that make program optimization possible. It explains the compiler optimization options and parameters and describes in detail all of the types of optimization that the compilers can perform. It provides a set of programming techniques that you can use to improve the optimizer's performance, and it explains the procedures for debugging optimized code.

Introduction to Compiler Technology

The Concurrent Computer Corporation's compilers for the Ada, C, C++, and Fortran programming languages are based on the Common Code Generator (CCG) technology. It is this technology that makes it possible to provide source-level compatibility across multiple architectures, a key component of Concurrent's P^3I policy. One of the major focuses of CCG is to produce the highest quality code possible so that your application attains the highest performance possible. Part of this process is performing *optimizations*--that is, transformations of your code so that it does the same work either by taking less time or by using fewer machine resources.

Many of the optimizations are complex and interrelated. It is not always possible for an optimizer to determine the best form in which to express code; therefore, CCG compilers provide a wide range of options and parameters to help you obtain the best performance from your application.

One of the major features of the CCG optimizer is that it strives to ensure that optimizations are *profitable*--that is, that the optimized program runs at least as fast as the original, if not faster. That the optimizer should do so may seem obvious, but it is not always easy. Many loop optimizations, for instance, are profitable only if the loop is executed many times once it is entered. Other optimizations depend on a favorable allocation of values to registers for their profitability.

One result of the concern with profitability is that the compiler may fail to perform an optimization because it cannot determine whether or not the change will be profitable. You may be able to assist the compiler in such cases by making slight changes in your program.

Optimization options and parameters are explained in "Compiler Optimization Options" on page 20-2. The classes of optimizations that the compilers perform are described in "Classes of Optimizations" on page 20-10. Programming techniques for optimization are presented in "Programming Techniques" on page 20-30. Procedures for debugging optimized programs are explained in "Debugging Optimized Code" on page 20-32.

It is assumed that you are familiar with the procedures for using one or more of the compilers. For information specific to a compiler, refer to the *HAPSE Reference Manual* and the **ada(1)** system manual page; the Concurrent *C Reference Manual* and the **hc(1)**

system manual page; or the *hf77 Fortran Reference Manual* and the **hf77(1)** system manual page.

Compiler Optimization Options

Compiler optimization options include the **-O** option, which enables you to set the compiler optimization level, and the **-Q** option, which enables you to control the optimizer's behavior. Procedures for using these options are explained in "Setting the Compiler Optimization Level" on page 20-2 and "Controlling Compiler Optimizations" on page 20-3, respectively. Each compiler has a verbose option that you can use to obtain information about the compilation and about optimization. This option is described in"Obtaining Optimization Messages" on page 20-10.

Setting the Compiler Optimization Level

The **-O***n* option allows you to select one of five levels of optimization. These levels are described as follows:

Level 0 — This level is called NONE; it performs only relatively simple optimizations and limits the register allocator to binding only a small number of global variables to registers (see **-Qhuge_heuristic)**. The NONE level is provided to compile extremely huge, usually machine-generated, programs rapidly.

Level 1 — This level is called MINIMAL; it performs only relatively simple optimizations. The MINIMAL level is provided for fast compilation.

Level 2 — This level is called GLOBAL; it selects more optimizations than MINIMAL. The GLOBAL level provides a compromise between compile speed and execution speed by placing limits on how much certain optimizations can do.

Level 3 — This level is called MAXIMAL; it sets the limits placed on the GLOBAL optimizations higher.

Level 4 — This level is called ULTIMATE; it sets time and space limits to extremely high levels.

MINIMAL is the default level if you do not specify the **-O** option. Compilations at the MAXIMAL level may take significantly longer than those at the MINIMAL level, but the generated code is usually significantly faster code. "Classes of Optimizations" on page 20-10 provides more detail on the optimizations that are included in each level.

Optimizations are also classified as *safe* or *unsafe.* An unsafe optimization may change the behavior of the program under certain boundary conditions whose occurrence is usually rare; for instance, if your program manipulates integer values that are close to the minimum or maximum possible integer values, then an unsafe optimization can cause those computations to overflow. By default, unsafe optimizations are enabled at the GLOBAL MAXIMAL, and ULTIMATE levels. You can disallow unsafe optimizations by using the **-Qopt_class** option (see "Controlling Compiler Optimizations" on page 20-3). In gen-

eral, you obtain less performance from your program if you disable the unsafe optimizations; you should disable them only if your program fails otherwise.

Optimizations that are potentially unsafe are identified in "Classes of Optimizations" on page 20-10. These currently include test replacement and some cases of algebraic simplification.

Controlling Compiler Optimizations

The **-Q***option-spec* option provides more precise control over the optimizer's behavior by allowing you to selectively enable or disable some of the optimizations described in "Classes of Optimizations" on page 20-10. In general, you want to use these forms of the **-Q** option only after you have analyzed your application thoroughly and have understood which parts are the most important to optimize. It is suggested that you use **analyze(1)** to perform this analysis. Use of this tool is described in "Identifying Performance Problems" on page 20-32.

Optimization--especially of very large programs--can often take a large amount of CPU time and memory. The compiler has built-in time and space limits to prevent it from using excessive time or space; however, these limits can be overridden by other forms of the **-Q***option-spec* option.

Forms of the **-Q** option that can be used for optimization are presented next.

-Qalias_array_elements_limit=*N*
: (C++ only) Limits the number of objects (variables, array elements, structure fields) in an array element that the alias analysis will tell the optimizer about. The default is 100. A value of zero indicates unlimited.

-Qalias_const_subscripts_limit=*N*
: (C++ only) Limits the number of array elements that the alias analysis will track. The default is 3. If your source uses a lot of constant subscripted array elements, increasing this option will allow the optimizer to treat those elements as separate variables. Setting this option to a high number or most programs, however, will just increase compile time without significantly increasing the precision of the alias analysis.

-Qalias_object_limit=*N*
: (C++ only) Limits the number of objects (variables, array elements, structure fields) that the alias analysis will tell the optimizer about. The default is 10,000. A value of zero indicates unlimited.

-Qalias_structure_fields_limit=*N*
: (C++ only) Limits the number of objects (variables, array elements, structure fields) contained in a given structure or union that the alias analysis will tell the optimizer about. The default is 100. A value of zero indicates unlimited.

-Qalias_object_limit=*N*
: (C++ only) Limits the number of objects (variables, array elements, structure fields) that the alias analysis will tell the optimizer about. The default is 10,000.

-Qalign_double=*N*

(Fortran only) Specifies the byte boundary to which `REAL*8`, `COMPLEX*8` and `COMPLEX*16` variables are aligned within common blocks.

Specifying **`-Qalign_double=8`** is equivalent to the default operation. Using the default **`-Qalign_double=8`** option, or **`-Qalign_double`**, aligns variables of these types to double-word boundaries. This eliminates having to manually align the variables.

Programs compiled with **`-Qalign_double=8`** may not be strictly "standard-conforming" because the standard does not permit gaps in common block layout. On the PowerPC, doubles aligned on a 4-byte boundary but not on an 8-byte boundary have a small execution-time penalty.

`-Qavoid_overflow`

(Fortran only) For some complicated operations, such as, dividing `COMPLEX` numbers or taking the absolute value of a `COMPLEX` number, the most straight-forward and efficient implementation can encounter overflow on intermediate results even though the final answer is representable. This is possible only if the real or imaginary portions are greater in magnitude than the square root of the largest real number (greater than about 1.844674e+19 for single precision and 1.340780793e+154 for double precision). The use of this option causes the compiler to generate slower code to avoid these overflows.

`-Qbenchmark`

Sets the optimization level to MAXIMAL; enables all unsafe optimizations; sets all time and space limits to extremely high values.

-Qblock_limit=*N*

(Fortran only) Limits the number of different `COMMON` blocks that will be treated as unique entities by the optimizer to *N*. The default is 128 at GLOBAL and 10,000 at MAXIMAL and ULTIMATE. Normally an assignment to a variable in one `COMMON` block does not affect variable and expression optimizations involving variables in other `COMMON` blocks. This may not be true if **`-Qblock_limit`** is exceeded. This option has no effect on `COMMON` blocks that do not exceed the limit specified by **`-Qvariable_limit`**.

-Qgrowth_limit=*N*

Limits the percent by which the optimizer is allowed to increase program size (for each subprogram) to *N*. *N* is an integer representing a percent; the default is 50 percent at GLOBAL, 200 percent at MAXIMAL and 10,000 percent at ULTIMATE for the supporting hardware platforms. This option controls several optimizations that replicate program code.

Keep in mind that the optimizer operates on an intermediate representation of the program; the size of this intermediate representation does not always accurately reflect the actual size of the generated code. Therefore, the percent that you specify for **`-Qgrowth_limit`** is only an approximation.

-Qhuge_heuristic=*N*

Limits the number of simultaneously alive global variables that the register allocator will attempt to bind to a register. This is very useful for compiling extremely huge, usually machine-generated, modules. The default is 33 at NONE and 1,000,000 (i.e., unlimited) otherwise.

`-Qignore_optimization_hints`

(C++ only) Directs the compiler to ignore optimization hint pragmas (See "Giving Hints to Compiler Optimizations (C++ only)" on page 20-8) embedded in the source.

`-Qinline=`*routine list*

(C only) Directs the compiler to treat the comma separated list of routines as though they had been specified as `inline` in the source. This will also work in C++ on routines with C linkage (`extern "C" {...}`).

`-Qinline_depth=`*N*

(C, C++ only) Limits the depth that inline functions will actually be inlined in other inline functions. Beyond that depth, out-of-line instances are invoked instead. The default is 1 for NONE and MINIMAL (meaning no inline expansion will happen inside a routine that is being expanded inline), 2 for GLOBAL, MAXIMAL, and ULTIMATE (meaning that routines can be inline expanded inside other routines that are inline expanded, but they in turn will not have routines inline expanded in them). Set the limit to higher numbers with caution as it can result in an huge increase in program size and hurt performance.

`-Qalias_object_limit=`*N*

(C++ only) Limits the number of objects (variables, array elements, structure fields) that the alias analysis will tell the optimizer about. The default is 10,000.

`-Qinvert_divides`

Hoists divides by region constants (an expression whose value will not change during the execution of the loop containing it) out of loops and replace them with a multiply by the reciprocal in the loop. In C and Ada, it also will transform divides by literals into multiplies by the reciprocal. (Fortran always does this unless **`-Qno_reciprocal_multiply`** is used.) This is the default for ULTIMATE.

`-Qflow_insensitive_alias_analysis`

(C++ only) Normally, the alias analysis takes into account whether a particular assignment to a pointer can actually reach a particular use of that pointer, i.e., makes use of the actual program flow. This option causes the alias analysis to assume all definitions of a pointer can reach all uses of it. Usually, this makes the analysis run a little faster at the expense of making some pessimal aliasing assumptions. Sometimes, however, this option will greatly increase compile time.

`-Qno_multiply_add`

Disables combining multiplies with adds in a single instruction.

`-Qloops=`*N*

Limits the number of loops for which the compiler will perform the copy-variable optimization to *N* (see "Copy Variables" on page 20-15). The default is 20 at GLOBAL and 1000 at MAXIMAL and ULTIMATE.

`-Qno_float_var_args`

(PowerPC only) Causes floating point registers to not be dumped to an array an the stack when **`var_args`** is used. Using this option causes the compiler

to not store the floating point registers. It should only be used if floating point arguments will never be passed to the `var_args` subroutine being compiled.

`-Qno_invert_divides`
(Fortran only) Disables the transformation of divide by a floating-point constant into multiply by the reciprocal of that constant.

`-Qno_multiply_add`
Disables combining multiplies with adds in a single instruction.

`-Qno_reciprocal_multiply`
(Fortran only) Disables the transformation of divide by floating-point constant into multiply by the reciprocal of that constant.

`-Qno_short_circuit`
(Fortran only) Do not short-circuit logical operations. The result of `.AND.` or `.OR.` may be known by evaluating only the first operand, i.e., (`.FALSE. .AND.` *anything*) is `.FALSE.`; (`.TRUE. .OR.` *anything*) is `.TRUE.` By default, the compiler may or may not short-circuit `.AND.` and `.OR.` logical operators depending on the estimated efficiency of the operations. Where the terms of a logical expression are scalar variable references and literals, the full logical expression is evaluated. In cases where a logical expression has more complicated operands with possible side effects, it is short-circuited. Therefore, short-circuit semantics are maintained unless the **`-Qno_short_circuit`** option is specified.

`-Qno_skew_large_arrays`
(Fortran only) Disables skewing large arrays. See **`-Qskew_large_arrays`**. This is the default at GLOBAL and MAXIMAL.

`-Qobjects=`*N*
Sets the number of variables that the compiler will optimize to *N*. (See "Variable Optimizations" on page 20-12.) The default is 128 at GLOBAL and 10,000 at MAXIMAL and ULTIMATE.

`-Qopt_class=`*setting*
Enables or disables unsafe optimizations according to the value of *setting*. The value of *setting* may be `safe, unsafe, or standard.` Specify either `safe` to disable unsafe optimizations or `unsafe` to enable them. Specify `standard` to enable unsafe optimizations specifically allowed by the language standard.

Individual compilers may allow additional values for setting. Refer to the appropriate language reference manual to determine the acceptable setting values, precise meanings and defaults.

`-Qoptimize_for_space`
Specifies that space rather than time is the critical factor in optimizing this program. Note that this option sets **`-Qgrowth_limit`** to zero.

`-Qpeel_limit_const=`*N*
Specifies the minimum number of iterations the loop unrolling algorithm will peel from a loop (see "Loop Unrolling and Software Pipelining" on page 20-22). This is used to achieve the effect of software pipelining so that each iteration of the resulting loop might overlap instructions from *N+1* iterations

of the original loop. The default is 1 at GLOBAL and 2 at MAXIMAL and ULTIMATE.

-Qpeel_var

Enables peeling a single iteration off a loop when the iteration count is unknown at compile time (i.e., is variable). This is used to achieve the effect of software pipelining so that each iteration of the resulting loop might overlap instructions from 2 iterations of the original loop. This is done by moving instructions from the loop into the loop's preheader, and moving the corresponding instruction from the peeled iteration into the loop. Thus the preheader primes the software pipeline, and the remaining instructions in the peeled iteration drain it. Because this can adversely effect cache behavior in loops that execute only a few times, this optimization is off by default.

-Qprecise_alias

(C++ only) Directs the alias analysis to perform precise alias analysis. This is default for GLOBAL, MAXIMAL, and ULTIMATE. See also **-Qquick_alias**.

-Qquick_alias

(C++ only) Directs the alias analysis to quickly make worst case assumptions about everything. Elements of arrays and fields of structures are not dealt with as separate objects. Any local variable whose address is taken is assumed to be aliased by all pointer indirections and function calls. This is default for NONE and MINIMAL. See also **-Qprecise_alias**.

-Qskew_large_arrays

(Fortran only) Skew the start of large local arrays onto unique data cache sets to prevent primary cache collisions. Membership in a primary cache set depends on the memory address modulo page size, which is further subdivided modulo cache line size. This is the default for ULTIMATE.
Thus, data at similar page offsets cannot occupy the same primary cache line at the same time. By aligning the start of large arrays to unique cache sets, array elements with similar indices such as `X(I)` and `Y(I)` do not occupy the same data cache line and may be co-resident in the cache, improving cache hit frequency for proximate references. On the PowerPC, cache lines are assumed to be 128 bytes and that 512 such cache lines fit into the 64KB cache.

A large array is considered larger than the primary cache. Note that this skewing applies only to uninitialized, non-equivalencies, non-character local arrays. For further information see the "Cache and Bus Interface Unit Operation" chapter in the *PowerPC 604 RISC Microprocessor User's Manual*.

-Qunroll_limit=*N*

Limits the number of times a loop with an iteration count that is a compile-time constant may be unrolled to *N*. (see "Loop Unrolling and Software Pipelining" on page 20-22). The default is 1 at GLOBAL and 10 at MAXIMAL and ULTIMATE.

The resulting code consists of the unrolled loop plus zero or more remainder iterations that are placed immediately after the unrolled loop. See also **-Qpeel_limit_const** to control the number of iterations in the remainder portion of the unrolled code.

`-Qunroll_limit_var=`*N*

Limits the number of times a loop with an iteration count that is not a compile-time constant (i.e., that is variable) may be unrolled to *N*. (see "Loop Unrolling and Software Pipelining" on page 20-22). The default is 1.

The resulting code consists of the unrolled loop plus a cleanup loop for the remainder iterations. If the loop is unrolled twice, the cleanup "loop" executes at most once, and so is not a loop. This option is disabled by **`-Qpeel_var.`**

`-Qvariable_limit=`*N*

(Fortran only) Limits the number of variables in each `COMMON` block that will be treated as unique entities by the optimizer to *N*. The default is 128. Normally an assignment to a variable in a `COMMON` block does not affect variable and expression optimizations involving other variables in that `COMMON` block. This may not be true if **`-Qvariable_limit`** is exceeded.

These options are explained in more detail in "Classes of Optimizations" on page 20-10.

If you have an application about which you know very little and you want to try to obtain the maximum performance from it, enable the **`-O4`** option. Specifying **`-O4`** removes all safety limits on compile time and space; hence, you should use it only when plenty of CPU and memory resources are available. You can reimpose limits removed by **`-O4`** by specifying other **`-Q`** options after the **`-O4`** specification.

Giving Hints to Compiler Optimizations (C++ only)

The alias analysis phase of the C++ compiler may be given several hints with `#pragma`s embedded in the source. These allow the user to specify information that can normally only be obtained through interprocedural analysis. Use them with caution, as incorrect hints can cause invalid optimizations to occur in later optimization phases. To specify a routine in these pragmas, an entire signature must be used. For example,

```
#pragma never_returns void print_error_and_exit(int, char *)
```

Variable lists are comma separated names of variables (with scoping operators as needed) and may be an empty list.

`#pragma nonrecursive` *routine-signature*

Tells the compiler that calling the designated routine will not result in the caller routine being called, i.e., will not result in recursion. Further, if designated routine is the routine being defined, it means that no routine called by the designated routine will result in itself being called recursively.

The effect of this pragma is to let the alias analysis and optimizer know that local static variables will not be modified by function calls unless their address as been made visible to other routines.

`#pragma explicit_use_def` *routine-signature*

Tells the compiler that no variable visible to the caller routine will be used or modified (defined) by calling the designated routine, unless there are exceptions listed in subsequent `#pragmas` (see `maybe_use`, `maybe_def`, and `definitely_def` below).

Ordinarily, the optimizer must assume function calls kill every externally visible variable, a worst case assumption that is rarely true.

`#pragma maybe_use` *routine-signature* {*variable-list*}*[*`,parameters`*][*`,all`*]*
Tells the compiler that the specified, comma separated, list of variables are the only variables visible to the caller whose values are referenced by the designated routine. Using this pragma implies the `explicit_use_def` pragma. This pragma may be used several times on the same designated routine: the effects are cumulative.

The optional `,parameters` designation tells the compiler that objects pointed to by pointer parameters may also be referenced. Note that this does not apply to objects pointed to by pointers contained in objects referenced. If you pass a pointer to a node in a linked list and use the `,parameters` designation, the compiler will assume the fields of that node are referenced, but not other objects pointed to by fields of that node.

The optional `,all` designation directs the compiler to make worst case assumptions about what the designated routine might reference. This may not be combined with a variable list or the `,parameters` designation.

`#pragma maybe_def` *routine-signature* {*variable-list*}*[*`,parameters`*][*`,all`*]*
Tells the compiler what variables visible to the caller might be defined by calling the designated routine. The optimizer will assume after the call that the designated variables might have whatever values they had before the call, or a new value given them by the call. As for `maybe_use`, this pragma will imply the `explicit_use_def` pragma, and also may be used multiple times to build up a longer list of variables.

The optional `,parameters` and `,all` designations operate the same way they do for the `maybe_use` pragma.

`#pragma definitely_def` *routine-signature* {*variable-list*}
Tells the compiler what variables visible to the caller will definitely be given a new value by calling the designated routine. As for `maybe_use`, this pragma will imply the `explicit_use_def` pragma, and also may be used multiple times to build up a longer list of variables.

`#pragma returns_new_object` *functon-signature*
Tells the compiler that the object pointed to by the pointer return value of the designated function is an uninitialized object that is newly allocated. Do not use this pragma on functions that return pointers to initialized structures, unions, or variables.

`#pragma returns_new_zeroed_object` *function-signature*
Tells the compiler that the object pointed to by the pointer return value of the designated function is newly allocated and all its bits have been set to zero.

`#pragma never_returns` *routine-signature*
Tells the compiler that the designated routine will never return. This gives the compiler more accurate flow information.

`#pragma pure_function` *function-signature*
Tells the compiler that the designated function neither uses nor modifies any variable that is visible to the caller and that it computes its return value

entirely from its actual arguments in a deterministic manner. This means that the compiler can eliminate the call if its result isn't used or if its result is redundantly computed elsewhere in the caller routine (common subexpression elimination). If an actual argument is of pointer type, it is implied that the result is computed by manipulating the actual bits of the pointer value, not by referencing the object pointed to by the pointer.

Obtaining Optimization Messages

Each compiler has a verbose (**-v**) option that produces output giving you more information about the compilation. Part of this output may include informative messages about optimization.

These messages inform you when the optimizer has been unable to perform one or more optimizations because of the limits in effect for the compilation. You can usually correct the problem by using the **-Q** option to specify a higher limit. Refer to the appropriate language reference manual or compiler man page to learn how to specify the verbose option.

Classes of Optimizations

CCG compilers perform the following classes of optimizations:

Branch Optimizations

The compiler performs branch optimizations to minimize the number of branches in the program and to reduce memory requirements. These optimizations include the following:

- Straightening blocks
- Folding conditional tests
- Eliminating unreachable code

- Inserting zero trip tests
- Duplicating partially-constant conditional branches

Each of these optimizations is described in the sections that follow. All are performed at the GLOBAL, MAXIMAL, and ULTIMATE levels.

Straightening Blocks

If two sections of code are executed in sequence, the optimizer rearranges the blocks to place them in sequence in the program so that it is not necessary to branch from one section to the other. Subprograms with very complicated flow of control (especially those using many `GOTO` statements) benefit most from this optimization.

Folding Conditional Tests

If all of the operands of a conditional test are constant, then the test can be replaced by a branch to the appropriate location. Programmers seldom intentionally write programs with such conditional tests. Most of the time, opportunities for this type of optimization arise as a result of other optimizations. Constant propagation often makes all the operands of conditional tests become constant (see “Copy Propagation” on page 20-14 for a description of this optimization). Inline expansion of a subprogram may also create opportunities for this type of optimization--especially if one or more arguments in the expanded call are constant values (see “Instruction Scheduling” on page 20-24 for a description of this optimization). Using macros in C also frequently generates opportunities for this optimization.

Eliminating Unreachable Code

The compiler eliminates code that it determines can never be executed. Code that cannot be executed is called *unreachable*. Code most often becomes unreachable as a result of folding a conditional test. Unreachable code usually results from programs that have a long history of modification and maintenance--especially in large and complicated subprograms, or from folding conditional tests.

Inserting Zero Trip Tests

To minimize the amount of branching within loops, the optimizer may insert *zero-trip tests* prior to the loop. This technique is used with loops that exit at the beginning of the loop rather than at the end. These “early exit” tests are duplicated before the loop; then the body of the loop is rearranged so that the test appears at the end of the loop. Inserting zero-trip tests also helps in such optimizations as code motion and strength reduction (see “Code Motion” on page 20-17 and “Strength Reduction” on page 20-20, respectively, for descriptions of these optimizations).

The **`-Qgrowth_limit`** option controls zero-trip test insertion. If the optimizer is unable to insert a zero-trip test because of the specified **`growth_limit`**, you may receive an informative message similar to the following:

```
foo.c, line 98: information: 50% growth limit prevents
    any more zero trip tests for this routine.
    See -Qgrowth_limit=N.
```

Duplicating Partially-Constant Conditional Branches

Another technique for minimizing branches is to duplicate conditional tests backward in the program to paths in which all of the operands of the test are assigned constant values. Constant propagation and folding conditional tests then replace the duplicated test with a direct branch (see “Copy Propagation” on page 20-14 and “Folding Conditional Tests” on page 20-11 for descriptions of these optimizations). On such paths of the program, then, no test is necessary.

The following Fortran fragment illustrates this type of optimization:

```
1.        IF(ETI.LT.0.0)ETI = 0.0
2.        IF(ETI.GT.1.0)GO TO 110
3.        IF(XFFINT)ETI = 10.0
4.  110   ...
```

If the program executes the assignment to `ETI` on line 1, the test on line 2 is obviously false. After optimization, this fragment is modified so that following the assignment on line 1, the program branches directly to line 3.

Variable Optimizations

For purposes of optimization, a *variable* is any scalar entity in the program that either has or can have a unique memory address. Not all of the variables that the compiler considers optimizing have names that you have declared, however, and some of the variables that you have declared may not be considered because they are not susceptible to optimization. Note the following:

- In some cases, an array element accessed by a constant subscript may be considered a variable.
- A scalar dummy argument in Fortran is usually considered a variable although it is passed by address.
- Scalar variables in large `COMMON` blocks may not always be considered variables.

CCG compilers perform the following optimizations on variables in your program:

- Dead code elimination
- Copy propagation
- Separate lifetimes
- Copy variables

Variable optimizations are performed only at the GLOBAL, MAXIMAL, and ULTIMATE levels. At the MAXIMAL and ULTIMATE levels, dead code elimination and copy propa-

gation are repeated several times. They are repeated because other optimizations can introduce additional opportunities for them; for instance, strength reduction may render a program variable unnecessary.

The number of variables that the compiler optimizes is limited by default. The optimizer chooses which of the variables in a subprogram to optimize according to the number of times that the variable is referenced. You can increase or decrease the number of variables that the optimizer will optimize by specifying the **`-Qobjects=`***N* option, where *N* represents the number of variables. Note, however, that this number may include some "artificial" variables created by the compiler as part of its translation of the source program.

If the verbose option is enabled and the optimizer observes more variables than the **`-Qobjects`** option allows it to optimize, the compiler issues an informative message such as the following:

```
foo.c, line 34: information: only first 128 most
    frequently occurring variables out of 337 total
    variables were optimized. See -Qobjects=N option.
```

Note that substantially increasing the value for the **`-Qobjects`** option may significantly increase compilation time and the amount of memory consumed by the compiler.

Each of the variable optimizations is described in the sections that follow.

Dead Code Elimination

An assignment to a variable that is not subsequently used or is always assigned another value before being used is called *dead code.* A set of assignments to a variable may also be dead if the values computed for the variable are used only in one or more of the assignments in the set. The following C fragment provides an example:

```
1.   i = 0 ;
2.   j = 0 ;
3.   while (j < 100) {
4.      i = i + 1 ;
5.      foo() ;
6.      i = i + 2 ;
7.      j = j + 1 ;
8.   }
```

The assignments to `i` (a local variable) on lines 4 and 6 compute values that are used only in those assignments (line 6 computes a value that will be used only on line 4, and vice versa). Those two assignments are actually dead code.

Most dead assignments occur because other optimizations have removed the uses of the variable. Strength reduction, for example, replaces some occurrences of an induction variable with compiler-generated temporary variables (see "Strength Reduction" on page 20-20 for a description of this optimization and a definition of *induction variable*). This procedure may cause the assignments to the induction variable to become dead code.

Dead code may also occur in large and complicated subprograms when new assignments are added or old code is removed. A new assignment may transform another assignment to the same variable into dead code. Removing old code may remove all of the uses for a particular assignment.

When debugging your program, you may notice that some assignments appear to be skipped or do not appear to have any associated code. Such discrepancies may be the result of dead code removal, or they may be caused by several other optimizations. See "Debugging Optimized Code" on page 20-32 for an explanation of the procedures for debugging optimized programs.

Copy Propagation

Copy propagation is an optimization in which an assignment to a variable is *propagated* to uses of that value of the variable. This propagation is performed by replacing references to the variable by the right-hand side of the assignment. In some cases, propagation allows the assignment to be removed; in other cases, it allows faster access to the value or reduces the usage of registers. There are three distinct types of copy propagation: constant, variable, and expression. Each type is explained in the paragraphs that follow.

Constant propagation is performed if the right-hand side of the assignment is a constant. In this type of propagation, the optimizer replaces as many references to the variable as it can. (It cannot, for example, replace a reference in which the address of the variable is used, as is the case when passing the variable by reference to another subprogram.) If all of the references that use the assigned value are replaced, then the assignment is removed.

A special form of constant propagation is performed for Fortran programs. A local variable (not in a `COMMON` block) that is initialized with a `DATA` statement and never modified in the subprogram can usually be replaced by the initializing constant. This form of constant propagation is performed primarily to accommodate older Fortran/66 programs in which `DATA` statements frequently substituted for the absence of a `PARAMETER` statement.

Variable propagation may be performed if the right-hand side of the assignment is another variable. Variable propagation is usually performed only if all references to the assigned value can be replaced and the assignment then removed. Even if the assignment cannot be removed, the optimizer may decide to perform the propagation if it determines that the variable on the right-hand side can usually be accessed faster than the variable on the left.

NOTE

In C, if the variable on the left-hand side of the assignment is declared `register`, variable propagation is not performed.

Expression propagation is attempted if the right-hand side of the assignment is an expression. To prevent unprofitable optimizations, the optimizer's use of this form of propagation has quite a few restrictions; for instance, only one reference to the assigned variable is replaced. If more than one reference uses the assigned value, the compiler refuses to propagate the expression. Furthermore, the replaced reference must not be inside a loop if the original assignment is not also in that loop.

Copy propagation may affect your debugging efforts even more than dead code removal. In addition to possibly removing assignment statements, copy propagation also affects the values of variables. Refer to "Debugging Optimized Code" on page 20-32 for additional information on this problem.

Separate Lifetimes

Using the same variable name for different purposes is fairly common practice. It often happens with loop-control variables; you may use the same variable to control two unrelated loops in a subprogram when you can as easily use two variables. The following Fortran program fragment provides an example:

```
1.      A = F(X)
2.      IF (A .GT. 0) THEN
3.         Y = A + B
4.      ELSE
5.         Y = B - A
6.      ENDIF
7.      A = G(X)
        ...
```

In this example, the references to `A` in lines 1-5 are distinct from the reference on line 7. You can use another variable name in the first set of lines without affecting the behavior of the program.

In these cases, the compiler makes each use of the variable a logically different variable (but maintains the same name). The separate variables can then be allocated to different locations (either to different registers or one instance to a register and the other to memory--the compiler never allocates separate lifetimes of a variable to two different memory locations). With this approach, a register is more likely to be available to hold the variable.

Copy Variables

In addition to the naturally occurring opportunities for separate lifetimes of variables, the optimizer creates more opportunities by inserting new assignments at strategic locations in the program. These assignments copy the variable to itself to introduce multiple separate lifetimes of the variable. Copy assignments for a variable used inside a loop, for example, may be inserted before and after the loop. In this way, the variable can be placed in a register for the duration of the loop, although outside the loop, a register is not available (or the variable must reside in memory for some other reason).

Copy variables are particularly effective with Fortran `COMMON` variables and Ada library-level package variables. Normally these variables must reside in memory so that other subprograms can access them; however, if a loop that uses such a variable does not call any other subprograms, then that variable can be allocated to a register during the loop.

Note that some variables are not subject to the copy-variables optimization; instead, they are treated as expressions (see “Expression Optimizations” on page 20-16 for a description of expression optimizations). Some examples of such variables are scalar Fortran dummy arguments and Ada variables that are declared in an enclosing subprogram.

The copy-variable optimization is restricted to the *N* most deeply nested loops in a routine, where *N* is specified with the **`-Qloops=`***N* option. If you have more than this number of loops in your program and the verbose option is enabled, you may receive an informative message similar to the following:

```
foo.c, line 34: information: copy variables applied only
    to first 20 most deeply nested loops out of 37 total
    loops. See -Qloops=N option.
```

Copy variables can affect debugging of optimized programs by making it difficult or impossible to examine the value of a variable. See “Debugging Optimized Code” on page 20-32 for an explanation of the procedures for debugging optimized programs.

Expression Optimizations

Expression optimizations refer to efforts made by the compiler either to eliminate the evaluation of an expression or to reduce the time or space required for that evaluation. The CCG optimizer applies the following expression optimizations:

- Algebraic simplification
- Address mode determination
- Common subexpression elimination
- Code motion

Algebraic simplification and address mode determination are always performed. Common subexpression elimination and code motion are performed only at the GLOBAL, MAXIMAL, and ULTIMATE levels. Each of these optimizations is explained in the sections that follow.

Algebraic Simplification

The compiler performs many transformations on expressions in order to eliminate unnecessary computations, take advantage of special hardware, and make optimum use of machine resources. The specific transformations performed vary from language to language and from one target architecture to another.

NOTE

The compiler does not perform an algebraic simplification if doing so violates the language’s rules concerning parentheses or the order in which expressions are evaluated.

Some of the transformations that are performed are described as follows:

1. An operation in which all of the operands are constants is folded into a single constant.
2. Constants within an expression are collected whenever possible by applying the laws of commutation, association, and distribution to the operations of addition, subtraction, multiplication, and division. This transformation creates additional opportunities for constant-folding.

Except for special cases, these transformations are limited to integer expressions to prevent introduction of unwanted round-off errors in floating-point operations. Even for integer operations, however, some of the transformations can be unsafe because of possible overflow. These transformations are enabled only if unsafe optimizations are allowed.

3. Arithmetic identity operations (for example, multiplying by zero or one and adding zero) are eliminated for both integer and floating-point operands.

4. Constants are factored out of integer expressions when possible; for example, the expression `(A*5)+(B*5)` is transformed into `(A+B)*5`. This transformation is performed only if unsafe optimizations are allowed.

5. Whenever possible, additive constants that appear in address computations are collected (for example, accessing array element `A(I+1)`). If the base address of the item being accessed is also constant, the additive constants are combined with it, thus eliminating one or more addition operations. If the base address is not constant, then the compiler attempts to rearrange the computation so that the addition can be performed by the addressing hardware.

6. For Fortran, some trigonometric and transcendental identities are also applied to expressions; for instance, `SIN(X) * COS(X)` is transformed into `0.5*SIN(2*X)`.

Address Mode Determination

System processors have the capability of combining the computation of an array-element address with the access to memory. Using these complex *address modes* can improve performance by reducing the amount of explicit computation required to access data. CCG compilers take advantage of this capability by analyzing address computations and by selecting the best address mode to use in each case.

Common Subexpression Elimination

Common subexpression elimination refers to the optimizer's attempt to avoid evaluating an expression whose value has already been computed. The optimizer analyzes each subprogram to determine the flow of data and the occurrence of each unique expression. If an expression is evaluated at a point where its value has previously been computed, the first evaluation saves the value, and the subsequent evaluation only references it. If there are some code paths to the point of evaluation that evaluate the expression and some paths that do not, the optimizer may insert computations of the expression on the paths where it is missing.

Code Motion

An expression that is computed inside a loop and whose value does not change within that loop is a candidate for *code motion*. The optimizer inserts a computation of the expression before entering the loop and saves that value. The computation within the loop is replaced with a reference to the saved value.

Code motion can sometimes be applied to an expression whose value does change within the loop. Consider the following Fortran program fragment:

```
1.       DO 10 I = 1,N
2.          IF (I .GT. M) THEN
3.             A = A - 2
4.          ELSE
5.             C = C + 1
6.          ENDIF
7.          X(I) = A + B
8.    10 CONTINUE
```

In this example, the value of `A + B` can be computed outside the loop and recomputed only when `A`'s value changes on line 3.

If you are programming in Ada, note that code motion may affect the behavior of the program if an expression raises a predefined exception such as `NUMERIC_ERROR`; for instance, although an expression may appear after an assignment to a variable, code motion may cause the expression to be evaluated before the assignment. Therefore, your program should not depend on this ordering unless the assignment and expression occur in different exception frames. (Code motion does not move an expression evaluation outside of any exception frame in which it occurs.)

Loop Optimizations

Because most programs spend the majority of their execution time in one or more loops, CCG provides an extensive set of loop optimizations. These optimizations are performed only at the MAXIMAL and ULTIMATE levels because they may significantly increase compile time. They may also significantly increase the amount of memory that your program requires, so you may need to use the **`-Qgrowth_limit`** option to control their behavior more precisely.

NOTE

The optimizer does not restrict its attention to loops formed by using high-level language constructs. Loops formed from conditional tests and explicit branches are also considered in loop optimizations.

Loop optimizations cannot be applied to loops with multiple entry points. Procedures for identifying such loops are explained in "Loops with Multiple Entry Points" on page 20-19.

The following optimizations are applied to loops:

- Strength reduction (See "Strength Reduction" on page 20-20.)
- Test replacement (See "Test Replacement" on page 20-21.)
- Duplicating loop exit tests (See "Duplicating Loop Exit Tests" on page 20-21.)

- Loop unrolling (See “Loop Unrolling and Software Pipelining” on page 20-22.)

Loops with Multiple Entry Points

Loop optimizations cannot be applied to loops with more than one point of entry. If the verbose option is enabled, the compiler warns you about such loops and attempts to transform them into single-entry loops by duplicating part of the loop body. The following Fortran procedure, for example, contains a loop with multiple entries:

```
1.       subroutine irred (arr,n)
2.       integer        arr(n)
3.       i = n - 1
4.       goto (10,20,30), i
5.    10 continue
6.       arr(i) = arr(n) - arr(i+1)
7.    20 continue
8.       arr (i+1) = arr(i) + arr(n)
9.    30 continue
10.      i = i - 1
11.      if (i .gt. 0) goto 20
12.      end
```

The messages you receive may be similar to the following:

```
At irreducible.f:7: information: Forward branch into
    loop number 1 repaired: Routine grew to 114%.
At irreducible.f:9: information: Forward branch into
    loop number 1 ends here and originates at line 4
```

Each loop is numbered internally by the compiler so that messages can refer to them uniquely. The messages provided in the example both refer to the same loop. The first message indicates whether or not the compiler has been able to repair the problem--if it has not, the message indicates why. This message also refers to the line in the source in which one of the entry points of the loop occurs--in this case, line 7. If the compiler has been able to repair the problem, this message tells you approximately how much more memory the transformed code occupies. In this example, the transformed code occupies about 14 percent more memory than the original.

The second message informs you where the second entry into the loop originates and terminates. This information enables you to modify your program to remove the problem. In this example, line 4 branches into the loop to line 9. If there are more than two entry points, the compiler repeats the second message for each one.

NOTE

You may occasionally see a message that refers to “unknown line.” Such a message means that the compiler cannot determine exactly which line has caused the extra entry point into the loop. It usually happens with programs that contain many `GOTO` statements.

The compiler repairs loops with multiple entries only if doing so does not violate the specified **growth_limit**. Note that the percent increase that the compiler reports is only an approximation because it is based on the compiler's internal representation of the program rather than the actual instructions generated. Furthermore, the compiler uses at most half of the allowed **growth_limit** in repairing these loops. If you specify **-Qgrowth_limit=30**, for example, repairing multiple-entry loops will increase program size by a maximum of 15 percent.

When the compiler repairs forward branches, it reports percent increases that are cumulative; that is, the amount of increase reflects the new total size of the procedure, including all previous repairs. Reporting a cumulative total helps you to select appropriate values for the **-Qgrowth_limit** option. It is important to note that the messages about repairs to multiple-entry loops are not necessarily generated in the same order in which the loops have been repaired. The new size reported in one message may be greater than that reported in a subsequent message. This indicates that the loops were repaired in a different order.

The following C program segment illustrates how you may unwittingly create a loop with multiple entry points by using a `goto`:

```
1.   if (a < b) {
2.      lab1:
3.      a += b ;
4.   }
5.   if (a == b) goto lab1 ;
```

This loop has two entry points because line 5 is part of the loop. When the test on line 1 is true, the loop is entered at label `lab1`. When the test on line 1 is false, the loop is entered at line 5.

Strength Reduction

Many of the loop optimizations involve the concepts of a region constant and an induction variable. A *region constant* is an expression whose value does not change within a loop. A variable is classified as an *induction variable* if all assignments to it within a loop have one of the following forms:

```
IV1 = IV1 + RC
IV1 = IV1 - RC
IV1 = IV2
IV1 = RC
```

where `RC` is a region-constant expression, and `IV1` and `IV2` are induction variables.

Strength reduction is an optimization that is applied to integer expressions in loops; it is applied to expressions that involve only addition and multiplication. One of the operands of the expression must be an induction variable; all of the other operands must be region-constant expressions. Such expressions can be reduced to simple addition operations that execute much faster.

Expressions to which strength reduction optimization can be applied occur more often than you may think. References to array elements that are indexed by an induction variable are usually candidates for strength reduction because the computation of the array-element address typically involves a multiplication (by the stride of the array) and one or more

addition operations. Multidimensional arrays usually involve more than one multiplication, so they benefit even more from strength reduction.

To ensure profitability, the optimizer performs strength reduction on an expression only if the expression is computed every time the loop body is executed. Expressions computed only inside an if-test in the loop, for instance, are not reduced. When possible, you should avoid writing loops in which a test for exiting the loop precedes other computations in the loop. Doing so prevents the optimizer from performing strength reduction on expressions appearing after the exit test. It may also prevent other useful optimizations from being performed.

Test Replacement

In many loops, an induction variable controls the number of iterations. The following C fragment provides an example:

```
for (i = 1; i < ending_value ; ++i) {
   ...
}
```

In this example, `i` is an induction variable whose value determines when the loop terminates. If there are one or more expressions involving `i` to which strength reduction can be applied and if the value of `i` is not required after the loop terminates, then the loop exit test can be modified to test the value of the reduced expression (the value to which `i` is compared is also suitably modified). This modification allows the variable `i` to be eliminated.

Although *test replacement* rarely causes a failure, it is potentially an unsafe optimization. If the induction variable used in the test can become large enough to cause one of the reduced expressions to overflow, then test replacement can cause a program to behave incorrectly. The problem most likely to occur is that the program loops infinitely. If you suspect that test replacement has caused a program to fail, disable unsafe optimizations, and recompile your program.

Duplicating Loop Exit Tests

At the MAXIMAL and ULTIMATE levels, the optimizer may duplicate a loop exit test elsewhere in the loop to avoid an unconditional branch. The following C fragment from a binary search algorithm illustrates the need for this optimization:

```
 1.   min = 0 ;
 2.   max = N - 1 ;
 3.   while (1) {
 4.      target = (min + max)/2 ;
 5.      if (arr [target] == elem) {
 6.         break ;  /* exit the loop, found */
 7.      } else if (arr [target] < elem) {
 8.         max = target - 1 ;
 9.      } else {
10.         min = target + 1 ;
11.      }
12.      if (min > max) break ;    /* exit the loop,
```

```
                                                    not found */
13.   }
```

Normally, after executing line 8, the program has to branch to line 12, where it tests whether to exit the loop. In this case, the optimizer may decide to duplicate the test on line 12 after line 8, thus eliminating an unconditional branch.

Internal limits and the **`-Qgrowth_limit`** option prevent this optimization from drastically increasing the size of the program. Because this optimization occurs after the other optimizations that are controlled by **`-Qgrowth_limit`**, the optimizer reserves 5 percent of the specified **`growth_limit`** for this optimization. If the preceding optimizations use less than 95 percent of the allowed **`growth_limit`**, this optimization is allowed to use all that remains.

If the **`growth_limit`** prevents a loop exit from being duplicated and the verbose option is enabled, you may see the following informative message:

```
foo.c, line 56: information: 25% growth limit prevents
    replacing unconditional branch with loop exit code.
    See -Qgrowth_limit=N.
```

Loop Unrolling and Software Pipelining

Unrolling a loop means that the loop body is duplicated one or more times, with the duplicates and the original body concatenated. The loop exit test, however, is not repeated for each duplication, so the unrolled loop executes one test for several executions of the loop body. This procedure reduces the overhead involved for each execution of the loop body and makes the loop run faster. Programs benefit more from loop unrolling because on pipelined and/or superscalar machines, computations from one copy of the loop may be overlapped with computations from another (see “Inline Expansion of Subprograms (Ada only)” on page 20-26 for a description of instruction scheduling).

Because the number of iterations may not be an integer multiple of the unrolling factor, there may be some clean up iterations following the unrolled loop. We refer to these as “peeled” iterations. **`reorder`** can take advantage of these peeled iterations to do an optimization called software pipelining. The basic idea of this optimization is to schedule some instructions from subsequent iterations during the current iteration. Some instructions from the unrolled body are moved into the block that branches to the loop, and the corresponding instructions in the peeled iterations are moved into the loop.

Loop unrolling is controlled by several of the **`-Q`** options.Loops with an iteration count that is known at compile-time are controlled separately from those with a variable iteration count. This is because unrolling a loop that iterates only a few times is often unprofitable. The compiler can make profitability decisions for the former, but the user must make them for the latter.

The **`-Qunroll_limit_const`** option specifies the maximum unroll factor for each loop whose iteration count is a compile time constant. A value of *N* means that the body of the loop is duplicated *N*-1 times to get a total of *N* copies of the body in the unrolled loop (thus, specifying a limit of one or zero disables this optimization). The optimizer determines the best unroll factor for each loop, but it never chooses a factor that exceeds **`unroll_limit`** or a factor that is greater than eight. the **`-Qpeel_limit_const`** option specifies a minimum number of times to be peeled off from the unrolled loop. This can be used to force software pipelining to be done even if the loop is not unrolled.

The **-Qunroll_limit_var** option specifies the unroll factor for loops with an iteration count that is not known at compile time (i.e., whose count is variable). Because the compiler does not know how many times the loop iterates, it also does not know how many iterations are peeled off. Thus there is a clean-up loop that is not unrolled after the unrolled loop. If the unroll factor is 2, this clean up loop is not actually a loop, but is a single iteration with a zero trip test before it. Software pipelining on these loops is done only if the **-Qpeel_var** option is used. This option turns off unrolling of those loops and peels a single iteration off. It is difficult, if not impossible, to predict when this is profitable.

The choice of unroll factor may be further limited by the **-Qgrowth_limit** option.When loop unrolling is performed, the amount of available growth that remains from previous optimizations minus approximately 5 percent (to allow for duplication of loop exit tests as explained in “Duplicating Loop Exit Tests” on page 20-21) is apportioned equally to all of the candidate loops. Thus, large loops may have a smaller unroll factor than small loops.

To be a candidate for loop unrolling, a loop must be controlled by an induction variable. If the initial value, increment, and final value of the induction variable are all constants, then the optimizer can determine exactly how many iterations the loop will perform. If possible, it chooses an unroll factor that evenly divides the total number of iterations. Otherwise, the unrolled loop is preceded by one or more copies of the loop body to make the number of iterations a multiple of the unroll factor.

If any one of the initial value, increment, or final value of the induction variable is not a constant, then the optimizer cannot replace the original loop. Instead, it constructs appropriate conditional tests that determine whether to execute the unrolled loop or the original. Furthermore, if the number of iterations is not a multiple of the unroll factor, the original loop may be executed after exiting the unrolled loop.

If the optimizer is unable to compute the total number of iterations, then unrolling the loop may gain little or nothing in performance. If, for instance, the loop is seldom executed or usually executed once, unrolling may actually degrade performance by adding additional overhead to the subprogram. You should probably disable loop unrolling for the subprogram.

The optimizer cannot choose the optimum unroll factor for loops with an unknown iteration count. In some cases, the unrolled loop may require more registers than are available, thus increasing memory accesses. As a result, the performance gain may be significantly less than you expect. Correction of these problems requires trial and error choice of the unroll limit and analysis of the program’s behavior.

In rare cases, loop unrolling may also worsen instruction-cache behavior by increasing the program size. If you suspect this is happening, disable loop unrolling, specify a very small **growth_limit**, or specify a smaller unroll limit.

If the specified **growth_limit** prevents the optimizer from unrolling a loop that otherwise can be unrolled and if you have enabled the verbose option on the compilation, you may receive one or more informative messages. The following message appears if the specified **growth_limit** prevents any loop from being unrolled:

```
foo.c, line 98: information: 25% growth limit prevents
    unrolling any loops in this routine.
    See -Qgrowth_limit=N.
```

If one or more loops are simply too large to be unrolled under the specified **`growth_limit`**, however, you may receive the following message for each of the loops:

```
foo.c, line 98: information: 25% growth limit prevents
    unrolling this loop. See -Qgrowth_limit=N.
```

Register Allocation

At all levels of optimization, the compiler performs sophisticated register allocation algorithms to make the best use of the machine registers and to minimize accesses to main memory. At the GLOBAL, MAXIMAL and ULTIMATE levels, however, the compiler performs more preliminary analysis of the program to provide even better register allocation.

The register allocator does not necessarily try to minimize the number of registers used; its goal is to minimize the amount of data movement between two registers or between registers and memory. Furthermore, the register allocator attempts to provide the instruction scheduler with more opportunities for rearranging instructions by evaluating expressions in different registers when possible.

Because of this approach, optimization may, in rare cases, cause a subroutine to execute more slowly than the un-optimized version. Slower execution results when some sections of the subroutine are rarely executed but require many registers for efficient execution. Those registers may have to be saved in memory when entering the subroutine and loaded again when exiting. Hence, the entry and exit code takes longer to execute, and the extra registers do not improve execution speed because the code in which they are used is seldom executed.

Instruction Scheduling

Instructions are divided into several classes. A different functional unit executes each class of instructions. As a result, several instructions can be executing simultaneously. The compilers take advantage of this capability by *scheduling*, or reordering, the instructions of the program and attempting to keep all of the functional units as busy as possible.

Instruction scheduling usually causes parts of several statements to be intermixed. You may be affected in two ways. First, instruction scheduling has effects similar to code motion when exceptions caused by evaluation of expressions occur (see “Code Motion” on page 20-17 for a description of code motion optimization). An expression that occurs after an assignment in the text may, in fact, be partially or completely evaluated before the assignment occurs. If that evaluation raises an exception, you cannot depend on the value of the variable to which the assignment is made. Note, however, that instruction scheduling obeys all of the rules of Ada so that an expression is never evaluated outside the exception frame in which it occurs.

Second, you may observe the effects of instruction scheduling when you are debugging the program; for instance, if you try to single step through the lines of the program, you may notice that the program seems to skip back and forth among two or more statements. The reason is that the instructions for those statements have been intermixed, yet each still carries with it the line number of the associated program text. Such information can be

invaluable if an exception occurs: once you find the offending instruction, you know exactly which line of your program has caused the failure. Unfortunately, debugging becomes somewhat more difficult.

When instructions are moved out of a basic block (eight linear set of instructions without branches), either to a place where it is being executed speculatively or to another block that always executes if the source block executes and vice versa. line number information is not carried along. Thus some parts of a statement might be executed long before debug information indicates.

By default, instruction scheduling is performed at the GLOBAL, MAXIMAL and ULTIMATE optimization levels. The C and Fortran compilers provide command-line options to disable instruction scheduling at the GLOBAL, MAXIMAL and ULTIMATE levels and enable it at the NONE and MINIMAL levels. For details, refer to the system manual pages for these compilers. Enabling instruction scheduling at MINIMAL is typically the cheapest compile-time method to get a significant performance boost.

Post-Linker Optimization

`analyze` optimizes programs during the post-linking stage. It uses program-wide, common subexpressions to optimize address and constant computation. (Refer to the **`analyze(1)`** man page for more information about **`analyze`**).

During the post-linking process, the compiler drivers pass the **`-O`** option to analyze, which invokes the post-linker optimization-code in **`analyze`**. This creates program-wide, common sub-expressions, and insures that the target instruction cache doesn't fail because of instruction misalignment.

Four reserved registers, `r28` through `r31` on the PowerPC, are set equal to the most common values that were loaded into registers using the "`lis rD,imm`" on the PowerPC. These values are usually the high-order, sixteen bits of the address of external variables. These same values get loaded repeatedly. By loading the reserved registers with the most common values at program start-up time, most loads and stores of external variables can be performed with one instruction instead of two instructions.

Additionally, if two different registers are loaded with the same value with `'lis'` instructions, and one of them reaches all of the uses of the other, **`analyze`** will substitute the former for the latter and eliminate the latter `'lis'` instruction even if its value isn't loaded into one of the registers.

The **`-W`** and **`-n`** options of **`analyze`** may be used to adjust the weighting of the static count of `lis` instructions.

The Concurrent compilation system puts additional relocation information into the vendor section to handle Fortran programs with assigned `GOTO` statements. Handwritten assembly code, or code produced by non-Concurrent compilers, might not be compatible with this optimization. The **`-X`** option can be used to exclude such routines.

If **`analyze`** detects a routine that references any of the reserved registers prior to optimization, **`analyze`** will generate a fatal error and refuse to optimize the program. Sometimes, certain assembly routines can reference these registers in a harmless fashion. The `setjmp` and `longjmp` routines, along with some signal handling code in `_sigtramp`, are known routines that are automatically excluded from optimization. Any other routines

that reference these registers can still be optimized by naming them with the **-X** option. This will cause **analyze** to ignore them.

Programs that use the threads library use register r31 as a process private data pointer (also called the *threads register*).

The link editor, **ld**, scans all object linked together, including both statically and dynamically linked libraries, and sets the ppdp_used flag in the vendor section if certain threads library routines are used. When **analyze** sees this flag set, it does not use register r31 to optimize lis instructions. See also the **thread(3thread)** man page.

Inline Expansion of Subprograms (Ada only)

The Ada compiler supports the substitution of subprogram bodies for subprogram calls. Such substitutions are controlled by user application of the predefined Ada language pragma, INLINE, and by inline configuration parameter limits.

The intent of pragma INLINE is to notify the compiler that particular subprograms should be considered for inline substitution, thereby eliminating the overhead associated with subprogram calls. Pragma INLINE can, therefore, be effective in maximizing performance while allowing the user to adhere to such higher level programming methodologies as modularity, data abstraction, and information hiding.

While the intent of pragma INLINE is to improve execution speed, there is no guarantee that the resultant code will actually run faster. In some cases, the overhead involved in the preservation of Ada language rules for subprogram calls (for example, copy-in/copy-out argument semantics, exception handling, and so on) may equal or even overshadow the savings achieved in removal of the actual subprogram call. Additionally, through repeated inline substitution within a single subprogram, the actual size of the subprogram may prevent other optimizations from occurring (for example, see the information on variable optimizations presented in “Variable Optimizations” on page 20-12). Pragma INLINE also creates additional compilation unit dependencies (as required by the Ada language), which cause additional routines to be recompiled after the body of a subprogram that has been expanded inline is modified. You can circumvent the overhead associated with implementing copy-in/copy-out semantics when the arguments on the subprogram call are constants or stack variables that are not visible to the body of the subprogram.

You can realize the most effective use of pragma INLINE by judiciously applying it in time-critical areas. Inline expansion is especially effective when it creates opportunities for other optimizations to occur; for instance, if a subprogram uses the value of an argument to select among various actions, and calls to the subprogram often pass a constant value for that argument, inline expansion, together with constant propagation, can eliminate the test and remove the unused actions (see “Copy Propagation” on page 20-14 for a description of constant propagation optimization). A subprogram called inside a loop is also a good candidate for inline expansion because it may allow code motion to move some of the expressions in the subprogram outside the loop; strength reduction may also be applied to the expressions in the subprogram (see “Code Motion” on page 20-17 and “Strength Reduction” on page 20-20 for descriptions of code motion and strength reduction optimizations, respectively).

The Ada compiler does not always honor the user's request to inline a subprogram call. The compiler issues a warning message when it rejects inline substitutions because of limitations on the form of subprograms or the form and type of subprogram arguments and when HAPSE inline configuration parameters are exceeded. Inline limitations and configuration parameters are described in the *HAPSE Reference Manual*.

Optimization of Constraints (Ada only)

The Ada programming language is more stringent concerning the integrity of data than such languages as C and Fortran. Ada declarations of variables and data types include the provision for specifying the values that are allowed for those entities. In many cases, the compiler must insert run-time tests to ensure that those constraints are obeyed; these tests are called *constraint checks*. They generally occur in one of the following contexts:

- An assignment to a variable may require a constraint check to ensure that the value being stored is valid.
- An operation such as addition may require a constraint check to ensure that the result is a valid value of the result's data type.
- An argument to a function or procedure may require a constraint check to ensure that the argument's value is within the range required by the formal parameter's data type.
- A dereference of an access variable may require a check to ensure that the access variable is not null.

At the GLOBAL, MAXIMAL and ULTIMATE levels, the CCG optimizer has the capability to remove these constraint checks when it can determine that they are unnecessary. As a simple example, consider the following Ada program fragment:

```
procedure doit is
   subtype little is integer range 1..10 ;
   a, b : little ;
   c : integer ;
begin
   ...
   c := a + b ;
end procedure doit ;
```

Normally, the addition `a + b` checks that its result does not exceed the bounds of an integer. In this case, however, the range of the operands precludes the possibility of overflow; therefore, the check can be removed.

Another example is provided by the following Ada program fragment:

```
procedure doit is
   subtype little is integer range 1..10 ;
   subtype bigger is integer range 1..100 ;
   a, b : little ;
   c : bigger ;
begin
   ...
```

```
   c := a + b ;
end procedure doit ;
```

The assignment to `c` normally requires a constraint check to ensure that the result of the addition is a valid value of type `bigger`. In this example, however, the types of `a` and `b` guarantee that their sum will be within the bounds of `bigger`; the constraint check is unnecessary.

The following Ada program fragment contains function calls:

```
package pkg is
    subtype little is integer range 1..10 ;
    subtype bigger is integer range 1..100 ;

    function fun1 (a, b: little) return bigger ;

    function fun2 (a, b: bigger) return bigger ;

    procedure doit is
        a, b : integer ;
        c : bigger ;
    begin
        ...
        c := fun1 (a, b) + fun2 (a, b) ;
    end procedure doit ;
end package pkg ;
```

The call to `fun1` imposes constraint checks on both `a` and `b`. Ordinarily, the call to `fun2` also imposes these constraint checks; however, the optimizer can remove these checks because they have been previously performed. Also note that the ranges of the operands of the addition imply that the result can never be smaller than two; thus, the assignment to `c` needs to check only the upper bound of its constraints.

The optimizer also uses comparisons in the program to narrow range restrictions on variables; for instance, in the following Ada program fragment:

```
procedure doit is
   subtype little is integer range 1..10 ;
   a, b : little ;
   c : integer ;
begin
   ...
   if c >= 1 then
      a := c ;
      if c <= 10 then
         b := c ;
      end if ;
   end if ;
end procedure doit ;
```

the assignment to `a` must check that the value of `c` does not exceed `10`. It does not have to check the lower bound because the if-test guarantees that `c` already meets that condition. Similarly, the assignment to `b` needs no constraint checks because the combination of the two if-tests guarantees that `c` lies in the range `1..10`.

In some cases, the range information derived from the program can be used to replace a variable with a constant. The following Ada program fragment provides an example:

```
procedure doit is
   subtype little is integer range 1..10 ;
   a, b : little ;
   c : integer ;
begin
   ...
   c := a ;
   if c < 2 then
      b := c ;
   end if ;
end procedure doit ;
```

After the assignment `c := a, c` is known to lie in the range `1..10`. Within the if-test, `c` is further restricted to be less than `2`; the only possible value for `c`, then, is `1`. In the assignment to `b`, therefore, the optimizer will replace `c` with the value `1`.

Obviously, the examples used here to explain the various types of constraint optimizations are very simple. Nevertheless, typical Ada applications benefit substantially from these optimizations.

Constraint optimizations are performed at the GLOBAL, MAXIMAL and ULTIMATE levels even when run-time constraint checks are suppressed by the user (either via the **`-S`** option or the predefined language pragma, `SUPPRESS`). As indicated in the preceding paragraphs, constraint optimizations benefit general code sequences and remove redundant constraint checks.

Inline Expansion of Subprograms (C++ only)

The C++ compiler supports the substitution of subroutine bodies for their calls. Such substitutions are controlled by use of the `inline` C++ keyword (which is implied in some contexts). For language specifics, the user is directed to any good C++ text.

The intent of `inline` is to notify the compiler that particular subroutines should be considered for inline substitution, thereby eliminating the overhead associated with subroutine calls. Inlining can, therefore, be effective in maximizing performance while allowing the user to adhere to such higher level programming methodologies as modularity, data abstraction, and information hiding.

While the intent `inline` is to improve execution speed, there is no guarantee that the resultant code will actually run faster. In some cases, the overhead involved in the preservation of C++ language rules for subroutine calls (for example, exception handling) may equal or even overshadow the savings achieved in removal of the actual subroutine call. Additionally, through repeated inline substitution within a single subroutine, the actual size of the subroutine may prevent other optimizations from occurring (for example, see the information on variable optimizations presented in “Variable Optimizations” on page 20-12).

You can realize the most effective use of `inline` by judiciously applying it in time-critical areas. Inline expansion is especially effective when it creates opportunities for other optimizations to occur; for instance, if a subroutine uses the value of an argument to select

among various actions, and calls to the subroutine often pass a constant value for that argument, inline expansion, together with constant propagation, can eliminate the test and remove the unused actions (see “Copy Propagation” on page 20-14 for a description of constant propagation optimization). A subroutine called inside a loop is also a good candidate for inline expansion because it may allow code motion to move some of the expressions in the subroutine outside the loop; strength reduction may also be applied to the expressions in the subroutine (see “Code Motion” on page 20-17 and “Strength Reduction” on page 20-20 for descriptions of code motion and strength reduction optimizations, respectively).

The C++ compiler does not always honor the user’s request to inline a subroutine call. In this case, an out-of-line instance is called instead.

Precise Alias Analysis (C++ Only)

All compilers do a certain amount of alias analysis to drive the optimization algorithms. Alias analysis determines what variables are being referred to by an expression such as `*p`, i.e., it determines what variables that expression is an alias for. Most compilers make simple worst case assumptions about aliasing, though some languages have more restrictive rules, such as the FORTRAN77 rule that a formal argument does not alias another formal argument or other variable visible to the subroutine.

The C++ does a more sophisticated analysis. It takes advantage of the assumptions allowed by the emerging C++ standard and also tracks assignments so that it has a more precise idea of the set of variables a pointer might be pointing too. Also, when the address of a variable is taken, it is possible to determine if that address gets passed to an external routine by way of a global variable or actual argument. If not, meaning that the address is used locally in a single subroutine only, it isn’t necessary to assume that the variable whose address was taken is killed by function calls.

This framework makes it easy and advantageous to add pragmas (See “Giving Hints to Compiler Optimizations (C++ only)” on page 20-8) to provide information to the optimizer about things that ordinarily could only be obtained by having an interprocedural optimizer analyzing the whole program and will be the enabling technology for future program analysis and debugging tools in the future.

Programming Techniques

The programming techniques that you can use for optimization of your code include coding techniques and performance analysis techniques. Coding tips are presented in “Coding Tips” on page 20-31. Performance analysis is discussed in “Identifying Performance Problems” on page 20-32.

Coding Tips

The CCG compilers are designed to obtain the highest performance code possible for your program, but they can go only so far in optimizing your program. It is recommended that you use the following techniques to improve a compiler's ability to optimize your program:

1. If you need to evaluate the same expression twice, write it exactly the same way each time. Do not write `a + b + c` one time and `a + c + b` the next.

2. Do not write loops with multiple entry points. Although the optimizer may be able to repair such loops, it may not be able to do so as well as you can.

3. Avoid writing loops that seldom execute more than once. If you cannot avoid writing such a loop, consider putting it in a separate subroutine so that any extra overhead imposed by optimizing the loop is confined to that routine. As an alternative, consider turning off loop unrolling for that routine.

4. If you are stepping through an array with a loop, try to make the *stride* a constant. The stride is the number of elements between successive elements examined by the loop. It is possible to make a loop with a stride that is not a constant although the increment of the loop counter is a constant. The following Fortran fragment provides an example:

```
      SUBROUTINE SUB(ARR,N,M)
      REAL ARR(M,N)
      DO 20 I=1,N
         ... ARR(1,I) ...
   20 CONTINUE
      END
```

   The reference to array `ARR` has a stride that is not a constant because Fortran arrays are stored in column-major order. Thus, each element of `ARR` that is accessed by the loop is `M` elements away from the last one accessed.

5. Traverse your data as compactly as possible to minimize paging and cache misses. This implies traversing Fortran column-order arrays from first index to last index, and C and Ada row-order arrays from last index to first index.

6. Avoid writing routines that contain a large amount of code but usually check a condition and exit. Large routines typically require that several registers be saved on entry and restored on exit. The overhead becomes significant if the routine does little else once it is entered.

   If you are programming in C, consider writing a macro to perform the checks, thus avoiding a subroutine call when the "early exit" is taken.

   If you are programming in Ada, consider putting the checks in another routine for which you specify `pragma INLINE`.

7. If possible, use a local variable instead of a global variable. Global variables are less susceptible to optimization. If a routine performs many operations on a global variable, consider using a temporary local variable for all

of the computations. Store the resulting value in the global variable only at the last possible moment.

If you are programming in C, avoid frequent accesses to data through global pointers. The optimizer must assume that these pointers can change each time a subroutine is called or memory is modified through any pointer. If the global pointer does not change, consider copying it to a local variable.

8. Excessively large routines are generally less susceptible to optimization than small ones. The more complicated the logic of a large routine, the less optimization is likely to improve its performance. You must simply use your best judgment in considering whether such a routine should be split into two or more routines.

9. Ada programmers should specify `pragma INLINE` only on relatively small, simple routines. If `pragma INLINE` is specified too often, the calling routines may become very large, thus limiting the amount of optimization performed.

Identifying Performance Problems

If you wish to obtain the highest possible performance from your program, use the **`analyze(1)`** tool to profile your program more accurately so that you can identify specific sections within routines where time is consumed. **`analyze`** can also give you a disassembly listing of a routine that includes information about how well the various functional units are being utilized. For additional information on the use of these tools, refer to the corresponding system manual pages.

You can use the information gained from using these tools to determine the routines on which to focus efforts to increase the performance of your program. In the context of optimization, make sure that those routines are receiving the full benefit of the optimizer. Verify that none of the optimizer's safety limits has been encountered during the compilation. Also check the code in these routines for any of the problems listed in "Coding Tips" on page 20-31.

Debugging Optimized Code

Successfully debugging optimized code requires that you understand optimization's effects on debugging. It may require that you examine your code to ensure that you have not violated assumptions that the language rules allow the optimizer to make. "Understanding Optimization's Effects on Debugging" on page 20-33 describes how debugging is affected by optimization. "Examining Your Program" on page 20-34 provides some tips for examining your program.

Understanding Optimization's Effects on Debugging

Throughout this chapter, aspects of optimization that can affect the debugging of your program have been pointed out. Note that, before trying to debug an optimized program, you should first make sure that the bug is not reproducible at MINIMAL optimization. You can count on the following when you are debugging an optimized program:

- You can examine the values of global variables and obtain the value that has last been stored in memory. If the program is not currently executing a loop in which a particular variable is modified, the value you obtain is, indeed, the correct one.
- Because of limitations in the format of debug information in executable files, the debugger expects a given variable to reside in one and only one location throughout a subprogram; yet if the variable has been copied, it may reside in different locations at different points in the subprogram (see "Copy Variables" on page 20-15 for a description of copy variables optimization). For global variables, the location that the debugger examines is usually the one in memory.
- The line number reported by the debugger is correct to the extent that some part of that line is being executed.

More detailed debugging usually requires that you obtain an assembly listing of the section of the program that you are debugging. Most debuggers have some capability for relating a specific instruction to the line in the program that has generated that instruction. If you are reasonably adept at reading assembly language, you can usually determine where the instructions for a particular line are located. You can probably also determine the registers used for each variable involved.

You cannot count on the following when you are debugging an optimized program:

- Setting a breakpoint on a given line may not stop the program before that line is executed; in fact, it may not stop the program at all because another copy of the line may exist elsewhere and the program will execute that instead.
- Printing the value of a local variable does not necessarily yield the correct value; for instance, if an assignment to variable `a` has been propagated and eliminated, you may see an outdated value when you print variable `a` although your program is about to evaluate an expression involving variable `a`. When you examine the results of an expression such as `(a + b)` and then examine variables `a` and `b`, you may be surprised to find that the values of the variables do not match the computed value.

  If the program has executed past the last use of a particular variable in the current routine, the variable's value may not exist anywhere. It may have been allocated to a register, and that register may have been reused for something else.
- On the supporting hardware platforms, floating-point exceptions are imprecise by default. See "Floating-Point Exceptions" on page 17-6 for details on how to make them precise or to disable them.

- An exception may not occur precisely on the instruction that has caused the fault, although it will usually be close by. The reasons include (1) variations in the time required for different instructions to execute and (2) the machine's ability to execute multiple instructions at the same time.

NOTE

The Ada compiler ensures that exceptions occur in the correct frame by preventing the overlapped execution of instructions from different frames. The supporting hardware platforms support precise exceptions so the exception will be on the correct instruction.

Debugging inlined routines has some special considerations. While the line number information will reflect the source of the inlined routine, the stack frame is still that of the calling routine since calling the inlined routine did not create a new stack frame. Thus when you go up a frame while in an inlined routine, you will not find you self at the call site of the inlined routine, but at the call site of the caller of the inlined routine. Also the fact that optimizations may move code out of the place where the inlined routine was originally located and scatter it in various places in the calling routine, can also result in seemingly inexplicable behavior from the debugger. Optimization can, in fact, make it seem that the inlined routine as completely disappeared when in fact its operations have just been folded into the operations of the calling routine.

Examining Your Program

You may compile a program without enabling optimization, successfully execute it, then recompile it with optimization enabled, and find that it fails. It is important to note that it may not be the optimizer that is causing the failure. You should first determine the subprogram in which the problem has occurred and then verify that you have not violated any of the rules of the language of which the optimizer takes advantage; for example, check your code to determine whether or not you are (1) depending on the ordering of execution of statements, (2) using an uninitialized variable that has not been assigned a value, or (3) omitting the `volatile` attribute on a variable that is modified asynchronously. Examples that show what can happen in the first and second instances are presented in the paragraphs that follow.

The example Ada code sequence that follows erroneously assumes that the variable `cycle` will be incremented at least once. In the presence of optimization, the evaluation of `x/y` may be moved outside of the loop and cause an exception to occur before the execution of the loop (Ada R.M. 11.6(3)).

```
procedure erroneous(x,y : in float ; z : in out float) is
   cycle : integer := 0 ;
begin
   loop
      cycle := cycle + 1 ;
      z := z - x / y ;
      exit when z < 0.0 ;
   end loop ;
   z := z / float(cycle) ;
```

```
exception
when numeric_error =>
   z := z / float(cycle) ;
   -- <<< -- erroneous assumption that cycle > 0
end erroneous ;
```

If you forget to initialize a variable before using its value, your program may behave differently with optimization turned on. Consider the following C example:

```
 1.  double foo(i, j)
 2.  int   i, j ;
 3.  {
 4.     double    a, b ;
 5.     int       k ;
 6.
 7.     for (k = i ; k < j ; ++k) {
 8.        a = b + 1.0 ;
 9.        b = i * a ;
10.     }
11.     return a + b ;
12.  }
```

The first time that line 8 is executed, variable `b` will not have a defined value. Furthermore, if `j >= i`, the loop body will not be executed, and neither `a` nor `b` will have a defined value when line 11 is executed. The value used will be the value last stored in the variable's location. Because optimization may cause `a` or `b` to be stored in a different location from the one in which it is stored without optimization, the value used in the computation can be different.

The optimizer can help you locate such problems as these. If you enable caution messages on your compilation (with the C compiler, you can do so by using the **`-n`** option), the optimizer will report variables that it finds are uninitialized. The preceding example produces the following messages:

```
"example1.c", line 11:  caution: Possibly un-initialized
    item <a> detected
"example1.c", line 11:  caution: Possibly un-initialized
    item <b> detected
"example1.c", line 8:  caution: Possibly un-initialized
    item <b> detected
```

The word `possibly` in these messages means that there is at least one path through the program that assigns a value to the given variable, but there are also one or more paths to that line that do not assign a value to the variable.

6
Formats

Replace with Part 6 tab

Part 6 - Formats

Part 6 Formats

21
Introduction to Formats

21
Introduction to Formats

Introduction

If you are writing programs that operate on other programs or if you require low-level knowledge of the system to perform debugging, then you should understand the formats supported by the software development environment.

This part of the manual describes these formats.

Chapter 22 (“Executable and Linking Format (ELF)”) describes the executable and linking format, ELF.

Chapter 23 (“tdesc Information”) discusses text description information, tdesc.

Chapter 24 (“DWARF Debugging Information Format”) describes the debugging information format, DWARF. It is primarily a reprint of the DWARF specification from UNIX International.

Chapter 25 (“DWARF Access Library (libdwarf)”) covers the libdwarf library that provides access to DWARF debugging and line number information.

22
Executable and Linking Format (ELF)

22
Executable and Linking Format (ELF)

Introduction

This chapter describes the executable and linking format (ELF) object files. The first section, “Program Linking” on page 22-3, focuses on how the format pertains to building programs. The second section, “Program Execution” on page 22-35, focuses on how the format pertains to loading programs. For background, see Chapter 4 (“Link Editor and Linking”). There are three main types of ELF object files.

Relocatable file Holds code and data suitable for linking with other object files to create an executable or a shared object file.

Executable file Holds a program suitable for execution; the file specifies how `exec()` creates a program’s process image.

Shared object file Holds code and data suitable for linking in two contexts. First, the link editor processes the shared object file with other relocatable and shared object files to create another object file. Second, the dynamic linker combines it with an executable file and other shared objects to create a process image.

Programs manipulate object files with the functions contained in the ELF access library, **`libelf`**. See the **`(3E)`** man pages, “ELF Library” on page 16-3, and “ELF Files” on page 16-17 for details.

File Format

As indicated, object files participate in program linking and program execution. For convenience and efficiency, the object file format provides parallel views of a file’s contents, reflecting the differing needs of these activities. Table 22-1 shows an object file’s organization.

Table 22-1. Object File Format

Linking View	Execution View
ELF header	ELF header
Program header table *optional*	Program header table
Section 1	Segment 1
. . .	
Section *n*	Segment 2
. . .	
. . .	. . .
Section header table	Section header table *optional*

An *ELF header* resides at the beginning and holds a "road map" describing the file's organization. *Sections* hold the bulk of object file information for the linking view: instructions, data, symbol table, relocation information, and so on. Descriptions of special sections appear in the first part of this chapter. The second part of this chapter discusses *segments* and the program execution view of the file.

A *program header table*, if present, tells the system how to create a process image. Files used to build a process image (execute a program) must have a program header table; relocatable files do not need one. A *section header table* contains information describing the file's sections. Every section has an entry in the table; each entry gives information such as the section name, the section size, and so forth. Files used during link editing must have a section header table; other object files may or may not have one.

Although Table 22-1 shows the program header table immediately after the ELF header, and the section header table following the sections, actual files may differ. Moreover, sections and segments have no specified order. Only the ELF header has a fixed position in the file.

Data Representation

As described here, the object file *format* supports various processors with 8-bit bytes and 32-bit architectures. Nevertheless, it is intended to be extensible to larger (or smaller) architectures. Object files, therefore, represent some control data with a machine-independent format, making it possible to identify object files and interpret their contents in a common way. Remaining data in an object file use the encoding of the target processor, regardless of the machine on which the file was created. See Table 22-2.

Table 22-2. 32-Bit Data Types

Name	Size	Alignment	Purpose
`Elf32_Addr`	4	4	Unsigned program address
`Elf32_Half`	2	2	Unsigned medium integer
`Elf32_Off`	4	4	Unsigned file offset
`Elf32_Sword`	4	4	Signed large integer
`Elf32_Word`	4	4	Unsigned large integer
unsigned char	1	1	Unsigned small integer

All data structures that the object file format defines follow the “natural” size and alignment guidelines for the relevant class. If necessary, data structures contain explicit padding to ensure 4-byte alignment for 4-byte objects, to force structure sizes to a multiple of 4, and so forth. Data also have suitable alignment from the beginning of the file. Thus, for example, a structure containing an `Elf32_Addr` member will be aligned on a 4-byte boundary within the file.

For portability reasons, ELF uses no bit-fields.

Program Linking

This section describes the object file information and system actions that create static program representations from relocatable files and shared objects.

ELF Header

Some object file control structures can grow, because the ELF header contains their actual sizes. If the object file format changes, a program may encounter control structures that are larger or smaller than expected. Programs might therefore ignore “extra” information. The treatment of “missing” information depends on context and will be specified when and if extensions are defined.

```
#define EI_NIDENT 16

typedef struct {
    unsigned char e_ident[EI_NIDENT];
    Elf32_Half e_type;
    Elf32_Half e_machine;
    Elf32_Word e_version;
    Elf32_Addr e_entry;
    Elf32_Off  e_phoff;
```

```
        Elf32_Off  e_shoff;
        Elf32_Word e_flags;
        Elf32_Half e_ehsize;
        Elf32_Half e_phentsize;
        Elf32_Half e_phnum;
        Elf32_Half e_shentsize;
        Elf32_Half e_shnum;
        Elf32_Half e_shstrndx;
    } Elf32_Ehdr;
```

`e_ident` The initial bytes mark the file as an object file and provide machine-independent data with which to decode and interpret the file's contents. Complete descriptions appear in "ELF Identification" on page 22-6.

`e_type` This member identifies the object file type.

Name	Value	Meaning
`ET_NONE`	0	No file type
`ET_REL`	1	Relocatable file
`ET_EXEC`	2	Executable file
`ET_DYN`	3	Shared object file
`ET_CORE`	4	Core file
`ET_LOPROC`	0xff00	Processor-specific
`ET_HIPROC`	0xffff	Processor-specific

Values from `ET_LOPROC` through `ET_HIPROC` (inclusive) are reserved for processor-specific semantics. Other values are reserved and will be assigned to new object file types as necessary.

`e_machine` This member's value specifies the required architecture for an individual file.

Name	Value	Meaning
`EM_NONE`	0	No machine
`EM_M32`	1	AT&T WE 32100™
`EM_SPARC`	2	SPARC®
`EM_386`	3	Intel 80386™
`EM_68K`	4	Motorola 68000™
`EM_88K`	5	Motorola 88000™
`EM_860`	7	Intel 80860™

Name	Value	Meaning
EM_MIPS	8	MIPS R2000™
EM_S370	9	Amdahl™
EM_IBM	11	IBM® RS/6000™ & PowerPC™

Other values are reserved and will be assigned to new machines as necessary. Processor-specific ELF names use the machine name to distinguish them. For example, the flags mentioned in "ELF Header Flags" on page 22-9 use the prefix EF_; a flag named WIDGET for the EM_XYZ machine would be called EF_XYZ_WIDGET.

e_version — This member identifies the object file version.

Name	Value	Meaning
EV_NONE	0	Invalid version
EV_CURRENT	1	Current version

The value 1 signifies the original file format; extensions will create new versions with higher numbers. The value of EV_CURRENT, though given as 1 above, will change as necessary to reflect the current version number.

e_entry — This member gives the virtual address to which the system first transfers control, thus starting the process. If the file has no associated entry point, this member holds zero.

e_phoff — This member holds the program header table's file offset in bytes. If the file has no program header table, this member holds zero.

e_shoff — This member holds the section header table's file offset in bytes. If the file has no section header table, this member holds zero.

e_flags — This member holds processor-specific flags associated with the file. Flag names take the form EF_*machine*_flag. See "ELF Header Flags" on page 22-9 for flag definitions.

e_ehsize — This member holds the ELF header's size in bytes.

e_phentsize — This member holds the size in bytes of one entry in the file's program header table; all entries are the same size.

e_phnum — This member holds the number of entries in the program header table. Thus the product of e_phentsize and e_phnum gives the table's size in bytes. If a file has no program header table, e_phnum holds the value zero.

`e_shentsize` This member holds a section header's size in bytes. A section header is one entry in the section header table; all entries are the same size.

`e_shnum` This member holds the number of entries in the section header table. Thus the product of `e_shentsize` and `e_shnum` gives the section header table's size in bytes. If a file has no section header table, `e_shnum` holds the value zero.

`e_shstrndx` This member holds the section header table index of the entry associated with the section name string table. If the file has no section name string table, this member holds the value `SHN_UNDEF`. See "Section Header" on page 22-9 and "String Table" on page 22-22 for more information.

ELF Identification

As mentioned above, ELF provides an object file framework to support multiple processors, multiple data encodings, and multiple classes of machines. To support this object file family, the initial bytes of the file specify how to interpret the file, independent of the processor on which the inquiry is made and independent of the file's remaining contents. The initial bytes of an ELF header (and an object file) correspond to the `e_ident` member. See Table 22-3.

Table 22-3. e_ident[ ] Identification Indexes

Name	Value	Purpose
`EI_MAG0`	0	File identification
`EI_MAG1`	1	File identification
`EI_MAG2`	2	File identification
`EI_MAG3`	3	File identification
`EI_CLASS`	4	File class
`EI_DATA`	5	Data encoding
`EI_VERSION`	6	File version
`EI_PAD`	7	Start of padding bytes
`EI_NIDENT`	16	Size of `e_ident[]`

These indexes access bytes that hold the following values.

`EI_MAG0` to `EI_MAG3`
A file's first 4 bytes hold a "magic number," identifying the file as an ELF object file.

Name	Value	Position
ELFMAG0	0x7f	e_ident[EI_MAG0]
ELFMAG1	'E'	e_ident[EI_MAG1]
ELFMAG2	'L'	e_ident[EI_MAG2]
ELFMAG3	'F'	e_ident[EI_MAG3]

EI_CLASS

The next byte, e_ident[EI_CLASS], identifies the file's class, or capacity.

Name	Value	Position
ELFCLASSNONE	0	Invalid class
ELFCLASS32	1	32-bit objects
ELFCLASS64	2	64-bit objects

The file format is designed to be portable among machines of various sizes, without imposing the sizes of the largest machine on the smallest. Class ELFCLASS32 supports machines with files and virtual address spaces up to 4 gigabytes; it uses the basic types defined above.

Class ELFCLASS64 is reserved for 64-bit architectures. Its appearance here shows how the object file may change, but the 64-bit format is otherwise unspecified. Other classes will be defined as necessary, with different basic types and sizes for object file data.

EI_DATA

Byte e_ident[EI_DATA] specifies the data encoding of the processor-specific data in the object file. The following encodings are currently defined.

Name	Value	Meaning
ELFDATANONE	0	Invalid data encoding
ELFDATA2LSB	1	See below
ELFDATA2MSB	2	See below

More information on these encodings appears below. Other values are reserved and will be assigned to new encodings as necessary.

EI_VERSION

Byte e_ident[EI_VERSION] specifies the ELF header version number. Currently, this value must be EV_CURRENT, as explained above for e_version.

`EI_PAD` This value marks the beginning of the unused bytes in `e_ident`. These bytes are reserved and set to zero; programs that read object files should ignore them. The value of `EI_PAD` will change in the future if currently unused bytes are given meanings.

A file's data encoding specifies how to interpret the basic objects in a file. As described above, class `ELFCLASS32` files use objects that occupy 1, 2, and 4 bytes. Under the defined encodings, objects are represented as shown below. Byte numbers appear in the upper left corners.

Encoding `ELFDATA2LSB` specifies 2's complement values, with the least significant byte occupying the lowest address.

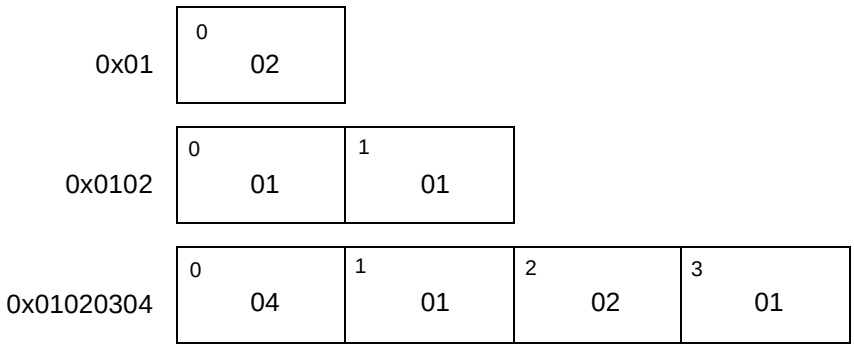


Figure 22-1. Data Encoding ELFDATA2LSB

Encoding `ELFDATA2MSB` specifies 2's complement values, with the most significant byte occupying the lowest address.

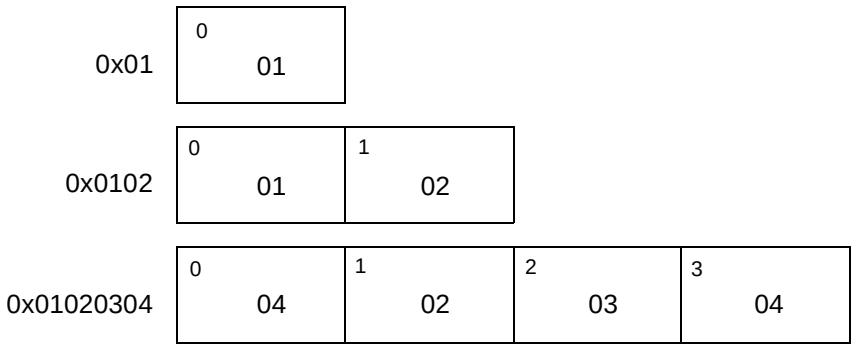


Figure 22-2. Data Encoding ELFDATA2MSB

ELF Header Flags

For file identification in `e_ident`, PowerUX uses the following values.

Table 22-4. PowerUX Identification, e_ident

Position	Value
`e_ident[EI_CLASS]`	`ELFCLASS32`
`e_ident[EI_DATA]`	`ELFDATA2MSB`

Processor identification resides in the ELF header's `e_machine` member and has the value `11`, defined as the name `EM_IBM`, or the value `5`, defined as the name `EM_88K`.

The ELF header's `e_flags` member holds bit flags associated with the file.

Table 22-5. Processor-Specific Flags, e_flags

Name	Value
`EF_PPC_SYSINUSER`	0x2
`EF_PPC_ADA`	0x40000000
`EF_PPC_ARMS`	0x80000000

`EF_PPC_SYSINUSER`
: This flag is defined by the 88open 88K ABI, but it is not presently used by PowerUX. If this flag is reset, it indicates that the application wishes full control of the layout of the virtual address space at addresses less than 0x80000000. If this flag is set, the operating system may place the stack and/or dynamic segments at lower addresses. This flag may be set for object files of type `ET_EXEC`. This flag shall not be set for object files of type `ET_REL` and `ET_DYN`.

`EF_PPC_ADA`
: The link editor sets this flag if the program was link edited with the **`-QAda`** option.

`EF_PPC_ARMS`
: The link editor sets this flag if the object was link edited with a map file that defined a Concurrent Ada ARMS segment.

Section Header

An object file's section header table lets one locate all the file's sections. The section header table is an array of `Elf32_Shdr` structures as described below. A section header table index is a subscript into this array. The ELF header's `e_shoff` member gives the byte offset from the beginning of the file to the section header table; `e_shnum` tells how many entries the section header table contains; `e_shentsize` gives the size in bytes of

each entry. Some section header table indexes are reserved; an object file will not have sections for these special indexes.

Table 22-6. Special Section Indexes

Name	Value
SHN_UNDEF	0
SHN_LORESERVE	0xff00
SHN_LOPROC	0xff00
SHN_HIPROC	0xff1f
SHN_ABS	0xfff1
SHN_COMMON	0xfff2
SHN_HIRESERVE	0xffff

SHN_UNDEF — This value marks an undefined, missing, irrelevant, or otherwise meaningless section reference. For example, a symbol "defined" relative to section number SHN_UNDEF is an undefined symbol.

Although index 0 is reserved as the undefined value, the section header table contains an entry for index 0. That is, if the e_shnum member of the ELF header says a file has 6 entries in the section header table, they have the indexes 0 through 5. The contents of the initial entry are specified later in this section.

SHN_LORESERVE — This value specifies the lower bound of the range of reserved indexes.

SHN_LOPROC through SHN_HIPROC — Values in this inclusive range are reserved for processor-specific semantics.

SHN_ABS — This value specifies absolute values for the corresponding reference. For example, symbols defined relative to section number SHN_ABS have absolute values and are not affected by relocation.

SHN_COMMON — Symbols defined relative to this section are common symbols, such as Fortran COMMON or unallocated C external variables.

SHN_HIRESERVE — This value specifies the upper bound of the range of reserved indexes. The system reserves indexes between SHN_LORESERVE and SHN_HIRESERVE, inclusive; the values do not reference the section header table. That is, the section header table does *not* contain entries for the reserved indexes.

Sections contain all information in an object file except the ELF header, the program header table, and the section header table. Moreover, object files' sections satisfy several conditions.

- Every section in an object file has exactly one section header describing it. Section headers may exist that do not have a section.
- Each section occupies one contiguous (possibly empty) sequence of bytes within a file.
- Sections in a file may not overlap. No byte in a file resides in more than one section.
- An object file may have inactive space. The various headers and the sections might not "cover" every byte in an object file. The contents of the inactive data are unspecified.

A section header has the following structure.

```
typedef struct {
    Elf32_Word  sh_name;
    Elf32_Word  sh_type;
    Elf32_Word  sh_flags;
    Elf32_Addr  sh_addr;
    Elf32_Off   sh_offset;
    Elf32_Word  sh_size;
    Elf32_Word  sh_link;
    Elf32_Word  sh_info;
    Elf32_Word  sh_addralign;
    Elf32_Word  sh_entsize;
} Elf32_Shdr;
```

`sh_name` This member specifies the name of the section. Its value is an index into the section header string table section (see "String Table" on page 22-22), giving the location of a null-terminated string.

`sh_type` This member categorizes the section's contents and semantics. Section types and their descriptions are listed in Table 22-7 and in the paragraphs for `SHT_SYMTAB` and `SHT_DYNSYN` immediately following Table 22-7.

`sh_flags` Sections support 1-bit flags that describe miscellaneous attributes. Flag definitions are given in Table 22-9.

`sh_addr` If the section will appear in the memory image of a process, this member gives the address at which the section's first byte should reside. Otherwise, the member contains 0.

`sh_offset` This member's value gives the byte offset from the beginning of the file to the first byte in the section. One section type, `SHT_NOBITS` described below, occupies no space in the file, and its `sh_offset` member locates the conceptual placement in the file.

`sh_size` This member gives the section's size in bytes. Unless the section type is `SHT_NOBITS`, the section occupies `sh_size` bytes in the file. A section of type `SHT_NOBITS` may have a non-zero size, but it occupies no space in the file.

`sh_link` This member holds a section header table index link, whose interpretation depends on the section type. Table 22-10 describes the values.

`sh_info` This member holds extra information, whose interpretation depends on the section type. Table 22-10 describes the values.

`sh_addralign` Some sections have address alignment constraints. For example, if a section holds a doubleword, the system must ensure doubleword alignment for the entire section. That is, the value of `sh_addr` must be congruent to 0, modulo the value of `sh_addralign`. Currently, only 0 and positive integral powers of two are allowed. Values 0 and 1 mean the section has no alignment constraints.

`sh_entsize` Some sections hold a table of fixed-size entries, such as a symbol table. For such a section, this member gives the size in bytes of each entry. The member contains 0 if the section does not hold a table of fixed-size entries.

A section header's `sh_type` member specifies the section's semantics.

Table 22-7. Section Types, sh_type

Name	Value
SHT_NULL	0
SHT_PROGBITS	1
SHT_SYMTAB	2
SHT_STRTAB	3
SHT_RELA	4
SHT_HASH	5
SHT_DYNAMIC	6
SHT_NOTE	7
SHT_NOBITS	8
SHT_REL	9
SHT_SHLIB	10
SHT_DYNSYM	11
SHT_LOPROC	0x70000000
SHT_HIPROC	0x7fffffff
SHT_LOUSER	0x80000000
SHT_VENDOR	0x80000000
SHT_HIUSER	0xffffffff

`SHT_NULL` This value marks the section header as inactive; it does not have an associated section. Other members of the section header have undefined values.

`SHT_PROGBITS` The section holds information defined by the program, whose format and meaning are determined solely by the program.

`SHT_SYMTAB` and `SHT_DYNSYM`
These sections hold a symbol table. Currently, an object file may have only one section of each type, but this restriction may be relaxed in the future. Typically, `SHT_SYMTAB` provides symbols for link editing, though it may also be used for dynamic linking. As a complete symbol table, it may contain many symbols unnecessary for dynamic linking. Consequently, an object file may also contain a `SHT_DYNSYM` section, which holds a minimal set of dynamic linking symbols, to save space. See "Symbol Table" on page 22-23 for details.

`SHT_STRTAB` The section holds a string table. An object file may have multiple string table sections. See "String Table" on page 22-22 for details.

`SHT_RELA` The section holds relocation entries with explicit addends, such as type `Elf32_Rela` for the 32-bit class of object files. An object file may have multiple relocation sections. See "Relocation" on page 22-27 for details.

`SHT_HASH` The section holds a symbol hash table. Currently, an object file may have only one hash table, but this restriction may be relaxed in the future. See "Hash Table" on page 22-59 for details.

`SHT_DYNAMIC` The section holds information for dynamic linking. Currently, an object file may have only one dynamic section, but this restriction may be relaxed in the future. See "Dynamic Section" on page 22-47 for details.

`SHT_NOTE` The section holds information that marks the file in some way. See "Note Section" on page 22-41 for details.

`SHT_NOBITS` A section of this type occupies no space in the file but otherwise resembles `SHT_PROGBITS`. Although this section contains no bytes, the `sh_offset` member contains the conceptual file offset.

`SHT_REL` The section holds relocation entries without explicit addends, such as type `Elf32_Rel` for the 32-bit class of object files. An object file may have multiple relocation sections. See "Relocation" on page 22-27 for details.

`SHT_SHLIB` This section type is reserved but has unspecified semantics.

`SHT_LOPROC` through `SHT_HIPROC`
Values in this inclusive range are reserved for processor-specific semantics.

`SHT_LOUSER` This value specifies the lower bound of the range of indexes reserved for application programs.

`SHT_HIUSER` This value specifies the upper bound of the range of indexes reserved for application programs. Section types between `SHT_LOUSER` and `SHT_HIUSER` may be used by the application, without conflicting with current or future system-defined section types. PowerUX reserves the low value, `SHT_VENDOR`, for vendor section information. See "Vendor Section" on page 22-18 for more information.

Other section type values are reserved. As mentioned before, the section header for index `0 (SHN_UNDEF)` exists, even though the index marks undefined section references. This entry holds the following.

Table 22-8. Section Header Table Entry: Index 0

Name	Value	Note
`sh_name`	0	No name
`sh_type`	`SHT_NULL`	Inactive
`sh_flags`	0	No flags
`sh_addr`	0	No address
`sh_offset`	0	No file offset
`sh_size`	0	No size
`sh_link`	`SHN_UNDEF`	No link information
`sh_info`	0	No auxiliary information
`sh_addralign`	0	No alignment
`sh_entsize`	0	No entries

A section header's `sh_flags` member holds 1-bit flags that describe the section's attributes. See Table 22-9 for defined values; other values are reserved.

Table 22-9. Section Attribute Flags, sh_flags

Name	Value
`SHF_WRITE`	0x1
`SHF_ALLOC`	0x2
`SHF_EXECINSTR`	0x4
`SHF_MASKPROC`	0xf0000000

If a flag bit is set in `sh_flags`, the attribute is "on" for the section. Otherwise, the attribute is "off" or does not apply. Undefined attributes are set to zero.

`SHF_WRITE` The section contains data that should be writable during process execution.

SHF_ALLOC	The section occupies memory during process execution. Some control sections do not reside in the memory image of an object file; this attribute is off for those sections.
SHF_EXECINSTR	The section contains executable machine instructions.
SHF_MASKPROC	All bits included in this mask are reserved for processor-specific semantics.

Two members in the section header, sh_link and sh_info, hold special information, depending on section type.

Table 22-10. sh_link and sh_info Interpretation

sh_type	sh_link	sh_info
SHT_DYNAMIC	The section header index of the string table used by entries in the section.	0
SHT_HASH	The section header index of the symbol table to which the hash table applies.	0
SHT_REL SHT_RELA	The section header index of the associated symbol table.	The section header index of the section to which the relocation applies.
SHT_SYMTAB SHT_DYNSYM	The section header index of the associated string table.	One greater than the symbol table index of the last local symbol (binding STB_LOCAL).
SHT_VENDOR	The section header index of the associated symbol table.	The section header index of the associated text section.
other	SHN_UNDEF	0

Special Sections

Various sections hold program and control information. Sections in the list below are used by the system and have the indicated types and attributes.

Table 22-11. Special Sections

Name	Type	Attributes
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.comment	SHT_PROGBITS	none
.data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.data1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.debug_abbrev	SHT_PROGBITS	none
.debug_arranges	SHT_PROGBITS	none
.debug_info	SHT_PROGBITS	none

Table 22-11. Special Sections (Cont.)

Name	Type	Attributes
.debug_line	SHT_PROGBITS	none
.debug_loc	SHT_PROGBITS	none
.debug_pubnames	SHT_PROGBITS	none
.dynamic	SHT_DYNAMIC	SHF_ALLOC + SHF_WRITE
.dynstr	SHT_STRTAB	SHF_ALLOC
.dynsym	SHT_DYNSYM	SHF_ALLOC
.fini	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.got	SHT_PROGBITS	see below
.hash	SHT_HASH	SHF_ALLOC
.init	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.interp	SHT_PROGBITS	none
.note	SHT_NOTE	none
.plt	SHT_PROGBITS	see below
.relname	SHT_REL	see below
.relaname	SHT_RELA	see below
.rodata	SHT_PROGBITS	SHF_ALLOC
.rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	none
.strtab	SHT_STRTAB	see below
.symtab	SHT_SYMTAB	see below
.tdesc	SHT_PROGBITS	SHF_ALLOC
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
$0001300	SHT_VENDOR	see below

.bss This section holds uninitialized data that contribute to the program's memory image. By definition, the system initializes the data with zeros when the program begins to run. The section occupies no file space, as indicated by the section type, SHT_NOBITS.

.comment This section holds version control information.

.data and **.data1** These sections hold initialized data that contribute to the program's memory image.

.debug_abbrev This section holds DWARF abbreviation tables.

.debug_arranges This section holds DWARF address ranges tables.

.debug_info This section holds DWARF debugging information entries.

.debug_line This section holds DWARF line number information.

.debug_loc This section holds DWARF location lists information.

.debug_pubname This section holds DWARF name lookup tables.

.dynamic This section holds dynamic linking information. See “Dynamic Linker” on page 22-46 for more information.

.dynstr This section holds strings needed for dynamic linking, most commonly the strings that represent the names associated with symbol table entries. See the section starting with “Dynamic Linker” on page 22-46 for more information.

.dynsym This section holds the dynamic linking symbol table. See “Symbol Table” on page 22-23.

.fini This section holds executable instructions that contribute to the process termination code. That is, when a program exits normally, the system arranges to execute the code in this section.

.got This section holds the global offset table. See “Global Offset Table” on page 22-54 for more information.

.hash This section holds a symbol hash table. See “Hash Table” on page 22-59 for more information.

.init This section holds executable instructions that contribute to the process initialization code. That is, when a program starts to run, the system arranges to execute the code in this section before calling the main program entry point (called `main` for C programs).

.interp This section holds the path name of a program interpreter. See “Program Interpreter” on page 22-45 for more information.

.note This section holds information as described in “Note Section” on page 22-41.

.plt This section holds the procedure linkage table. See “Procedure Linkage Table” on page 22-58 for more information.

.rel*name* and **.rela***name*
These sections hold relocation information, as “Relocation” on page 22-27 describes. If the file has a loadable segment that includes relocation, the sections’ attributes will include the `SHF_ALLOC` bit; otherwise, that bit will be off. Conventionally, *name* is supplied by the section to which the relocations apply. Thus a relocation section for **.text** normally would have the name **.rel.text** or **.rela.text**.

.rodata and **.rodata1**
These sections hold read-only data that typically contribute to a non-writable segment in the process image. See “Program Header” on page 22-35 for more information.

.shstrtab This section holds section names.

.strtab This section holds strings, most commonly the strings that represent the names associated with symbol table entries. If the file has a loadable segment that includes the symbol string table, the section's attributes will include the SHF_ALLOC bit; otherwise, that bit will be off.

.symtab This section holds a symbol table, as "Symbol Table" on page 22-23 describes. If the file has a loadable segment that includes the symbol table, the section's attributes will include the SHF_ALLOC bit; otherwise, that bit will be off.

.tdesc This section holds "text description" (tdesc) information. See Chapter 23 for more information.

.text This section holds the "text," or executable instructions, of a program.

$0001300 This section holds "vendor" information specific to applications built on PowerUX. See "Vendor Section" on page 22-18 for more information.

Section names with a dot (.) prefix are reserved for the system, although applications may use these sections if their existing meanings are satisfactory. Applications may use names without the prefix to avoid conflicts with system sections. The object file format lets one define sections not in the list above. An object file may have more than one section with the same name.

Vendor Section

A PowerUX-specific vendor section has the following structure.

```
struct {
    unsigned long magic;
    unsigned long text_reloc;
    unsigned char round_mode;
    unsigned char fp_except_kind;
    unsigned char float_exceptions;
    unsigned char IBM_mode;
    unsigned char float_precision;
    unsigned char ppdp_used;
    unsigned char fp_spec_exec;
    char           filler[21];
};
```

magic This member specifies the magic number. It is currently zero.

text_reloc This member provides the virtual address of relocation information pertaining to the "text" section, if there is any relocation information. This information is used by the **analyze(1)** utility.

round_mode This member specifies the IEEE floating-point rounding mode under which the program should begin execution. The various rounding modes and their values appear in Table 22-12.

`fp_except_kind` This member specifies whether floating-point exceptions interrupts should be enabled when the program begins execution and whether enabled exceptions should be precise or imprecise.

`float_exceptions` This member specifies the mask of the floating-point exceptions which should be enabled when the program begins execution. The various exceptions and their values appear in Table 22-14.

`IBM_mode` This member indicates whether the program uses any features unique to members of the PowerPC family. The various modes and their values appear in Table 22-15.

`float_precision` This member specifies whether or not the program uses IEEE, 80-bit floating-point precision. The values for this member appear in Table 22-16.

`ppdp_used` This member indicates whether the program uses the process private data pointer (i.e., register r31). This information is used by the **`analyze(1)`** utility. The values for this member appear in Table 22-16.

`fp_spec_exec` This member indicates whether the program contains floating-point code that is executed speculatively. This information is used by the **`ld(1)`** utility. The values for this member appear in Table 22-16.

Table 22-12. Vendor Section Rounding Modes, round_mode

Name	Value	Meaning
`_VND_RND_IEEENEAR`	0	Round to nearest
`_VND_RND_IEEEZERO`	1	Round to zero
`_VND_RND_IEEEPINF`	2	Round to positive infinity
`_VND_RND_IEEENINF`	3	Round to negative infinity
`_VND_RND_IEEECOMP`	4	This value indicates that the object file is compatible with object files of any other rounding mode. It is typically used in object files of system archives.

Table 22-13. Vendor Section Floating-Point Exceptions Kind, fp_except_kind

Name	Value	Meaning
`_VND_FPE_IMPRECISE`	0	Enable imprecise floating-point exceptions
`_VND_FPE_PRECISE`	1	Enable precise floating-point exceptions
`_VND_FPE_DISABLED`	2	Disable floating-point exceptions interrupt

Table 22-14. Vendor Section Enabled Exceptions, float_exceptions

Name	Value	Meaning
_VND_FPX_INV	16	Invalid operation
_VND_FPX_DZ	8	Divide-by-zero
_VND_FPX_UFL	4	Underflow
_VND_FPX_OFL	2	Overflow
_VND_FPX_IMP	1	Imprecise (inexact)

Table 22-15. Vendor Section PowerPC Features, IBM_mode

Name	Value	Meaning
_VND_MODE_POWERPC	0	This value indicates that the program does not contain features unique to any of the PowerPC architectures.
_VND_MODE_601	1	This value indicates that the program contains features unique to the PowerPC 601 architecture only.
_VND_MODE_603	2	This value indicates that the program contains features unique to the PowerPC 603 architecture only.
_VND_MODE_604	4	This value indicates that the program contains features unique to the PowerPC 604 architecture only.
_VND_MODE_620	8	This value indicates that the program contains features unique to the PowerPC 620 architecture only.
_VND_MODE_604_620	0xc	This value indicates that the program contains features unique to the PowerPC 604 and the 620 architectures only.
_VND_MODE_N603	0xd	This value indicates that the program contains features not on the PowerPC 603 architecture.
_VND_MODE_N601	0xe	This value indicates that the program contains features not on the PowerPC 601 architecture.

Table 22-15. Vendor Section PowerPC Features, IBM_mode

Name	Value	Meaning
_VND_MODE_POWERPCX	0xf	This value indicates that the program contains a mixture of features unique to particular PowerPC architectures.
_VND_MODE_604E	0x10	This value indicates that the program contains features unique to the PowerPC 604e architecture only.
_VND_MODE_601_604E	0x11	This value indicates that the program contains features unique to the PowerPC 601 and the 604e architectures only.

Table 22-16. Vendor Section Extended Double-Precision Use, float_precision

Name	Value	Meaning
_VND_FLOAT_NOT_EXT_DBL	0	Extended double-precision floating-point is not used
_VND_FLOAT_EXT_DBL	1	Extended double-precision floating-point is used.

Table 22-17. Vendor Section Process Private Data Pointer Use, ppdp_used

Name	Value	Meaning
_VND_PPDP_NOT_USED	0	Extended double precision floating-point is not used
_VND_PPDP_USED	1	Extended double-precision floating-point is used.

Table 22-18. Vendor Section FP Speculative Execution Use, fp_spec_exec

Name	Value	Meaning
_VND_FP_NOT_SPEC_EXEC	0	Floating-point speculative execution not done.
_VND_FP_SPEC_EXEC	1	Floating-point speculative execution done.

String Table

String table sections hold null-terminated character sequences, commonly called strings. The object file uses these strings to represent symbol and section names. One references a string as an index into the string table section. The first byte, which is index zero, is defined to hold a null character. Likewise, a string table's last byte is defined to hold a null character, ensuring null termination for all strings. A string whose index is zero specifies either no name or a null name, depending on the context. An empty string table section is permitted; its section header's `sh_size` member would contain zero. Non-zero indexes are invalid for an empty string table.

A section header's `sh_name` member holds an index into the section header string table section, as designated by the `e_shstrndx` member of the ELF header. The following figures show a string table with 25 bytes and the strings associated with various indexes.

Table 22-19. String Table

Index	+ 0	+ 1	+ 2	+ 3	+ 4	+ 4	+ 6	+ 7	+ 8	+ 9
0	\0	n	a	m	e	.	\0	V	a	r
10	i	a	b	l	e	\0	a	b	l	e
20	\0	\0	x	x	\0					

Table 22-20. String Table Indexes

Index	String
0	*none*
1	name.
7	Variable
11	able
16	able
24	*null string*

As the example shows, a string table index may refer to any byte in the section. A string may appear more than once; references to substrings may exist; and a single string may be referenced multiple times. Unreferenced strings also are allowed.

Symbol Table

An object file's symbol table holds information needed to locate and relocate a program's symbolic definitions and references. A symbol table index is a subscript into this array. Index 0 both designates the first entry in the table and serves as the undefined symbol index. The contents of the initial entry are specified later in this section.

Name	Value
STN_UNDEF	0

A symbol table entry has the following format.

```
typedef struct {
    Elf32_Word    st_name;
    Elf32_Addr    st_value;
    Elf32_Word    st_size;
    unsigned char st_info;
    unsigned char st_other;
    Elf32_Half    st_shndx;
} Elf32_Sym;
```

`st_name` This member holds an index into the object file's symbol string table, which holds the character representations of the symbol names. If the value is non-zero, it represents a string table index that gives the symbol name. Otherwise, the symbol table entry has no name.

`st_value` This member gives the value of the associated symbol. Depending on the context, this may be an absolute value, an address, and so forth; details appear below.

`st_size` Many symbols have associated sizes. For example, a data object's size is the number of bytes contained in the object. This member holds 0 if the symbol has no size or an unknown size.

`st_info` This member specifies the symbol's type and binding attributes. A list of the values and meanings appears in Table 22-21. The following code shows how to manipulate the values.

```
#define ELF32_ST_BIND(i)   ((i)>>4)
#define ELF32_ST_TYPE(i)   ((i)&0xf)
#define ELF32_ST_INFO(b,t) \
                (((b)<<4)+((t)&0xf))
```

`st_other` This member indicates whether or not the symbol was assembled with the **`-A`** option to **`as`**. A value of 0 indicates that the **`-A`**

option was not used. A value of 1 indicates that the **-A** option was used.

`st_shndx` Every symbol table entry is "defined" in relation to some section; this member holds the relevant section header table index. Some section indexes indicate special meanings.

A symbol's binding determines the linkage visibility and behavior.

Table 22-21. Symbol Binding, ELF32_ST_BIND

Name	Value
`STB_LOCAL`	0
`STB_GLOBAL`	1
`STB_WEAK`	2
`STB_LOPROC`	13
`STB_HIPROC`	15

`STB_LOCAL` Local symbols are not visible outside the object file containing their definition. Local symbols of the same name may exist in multiple files without interfering with each other.

`STB_GLOBAL` Global symbols are visible to all object files being combined. One file's definition of a global symbol will satisfy another file's undefined reference to the same global symbol.

`STB_WEAK` Weak symbols resemble global symbols, but their definitions have lower precedence.

`STB_LOPROC` through `STB_HIPROC`
Values in this inclusive range are reserved for processor-specific semantics.

Global and weak symbols differ in two major ways.

- When the link editor combines several relocatable object files, it does not allow multiple definitions of `STB_GLOBAL` symbols with the same name. On the other hand, if a defined global symbol exists, the appearance of a weak symbol with the same name will not cause an error. The link editor honors the global definition and ignores the weak ones.
- When the link editor searches archive libraries, it extracts archive members that contain definitions of undefined global symbols. The member's definition may be either a global or a weak symbol. The link editor does not extract archive members to resolve undefined weak symbols. Unresolved weak symbols have a zero value.

In each symbol table, all symbols with `STB_LOCAL` binding precede the weak and global symbols. As "Section Header" on page 22-9 describes, a symbol table section's `sh_info` section header member holds the symbol table index for the first non-local symbol.

A symbol's type provides a general classification for the associated entity.

Table 22-22. Symbol Types, ELF32_ST_TYPE

Name	Value
STT_NOTYPE	0
STT_OBJECT	1
STT_FUNC	2
STT_SECTION	3
STT_FILE	4
STT_LOPROC	13
STT_PPC_FCOMM	13
STT_HIPROC	15

STT_NOTYPE — The symbol's type is not specified.

STT_OBJECT — The symbol is associated with a data object, such as a variable, an array, and so forth.

STT_FUNC — The symbol is associated with a function or other executable code.

STT_SECTION — The symbol is associated with a section. Symbol table entries of this type exist primarily for relocation and normally have STB_LOCAL binding.

STT_FILE — Conventionally, the symbol's name gives the name of the source file associated with the object file. A file symbol has STB_LOCAL binding, its section index is SHN_ABS, and it precedes the other STB_LOCAL symbols for the file, if it is present.

STT_LOPROC through STT_HIPROC — Values in this inclusive range are reserved for processor-specific semantics.

STT_PPC_FCOMM — The symbol represents a Fortran COMMON block.

Function symbols (those with type STT_FUNC) in shared object files have special significance. When another object file references a function from a shared object, the link editor automatically creates a procedure linkage table entry for the referenced symbol. Shared object symbols with types other than STT_FUNC will not be referenced automatically through the procedure linkage table.

If a symbol's value refers to a specific location within a section, its section index member, st_shndx, holds an index into the section header table. As the section moves during relocation, the

symbol's value changes as well, and references to the symbol continue to "point" to the same location in the program. Some special section index values give other semantics.

`SHN_ABS` — The symbol has an absolute value that will not change because of relocation.

`SHN_COMMON` — The symbol labels a common block that has not yet been allocated. The symbol's value gives alignment constraints, similar to a section's `sh_addralign` member. That is, the link editor will allocate the storage for the symbol at an address that is a multiple of `st_value`. The symbol's size tells how many bytes are required.

`SHN_UNDEF` — This section table index means the symbol is undefined. When the link editor combines this object file with another that defines the indicated symbol, this file's references to the symbol will be linked to the actual definition.

As mentioned above, the symbol table entry for index 0 (`STN_UNDEF`) is reserved; it holds the following.

Table 22-23. Symbol Table Entry: Index 0

Name	Value	Note
`st_name`	0	No name
`st_value`	0	Zero value
`st_size`	0	No size
`st_info`	0	No type, local binding
`st_other`	0	(no 'other' information)
`st_shndx`	`SHN_UNDEF`	No section

Symbol Values

Symbol table entries for different object file types have slightly different interpretations for the `st_value` member.

- In relocatable files, `st_value` holds alignment constraints for a symbol whose section index is `SHN_COMMON`.
- In relocatable files, `st_value` holds a section offset for a defined symbol. That is, `st_value` is an offset from the beginning of the section that `st_shndx` identifies.
- In executable and shared object files, `st_value` holds a virtual address. To make these files' symbols more useful for the dynamic linker, the section offset (file interpretation) gives way to a virtual address (memory interpretation) for which the section number is irrelevant.

Although the symbol table values have similar meanings for different object files, the data allow efficient access by the appropriate programs.

Relocation

Relocation is the process of connecting symbolic references with symbolic definitions. For example, when a program calls a function, the associated call instruction must transfer control to the proper destination address at execution. In other words, relocatable files must have information that describes how to modify their section contents, thus allowing executable and shared object files to hold the right information for a process's program image. *Relocation entries* are these data.

```
deputed struct {
    Elf32_Addrr_offset;
    Elf32_Wordr_info;
} Elf32_Rel;

typedef struct {
    Elf32_Addr  r_offset;
    Elf32_Word  r_info;
    Elf32_Sword r_addend;
} Elf32_Rela;
```

`r_offset` This member gives the location at which to apply the relocation action. For a relocatable file, the value is the byte offset from the beginning of the section to the storage unit affected by the relocation. For an executable file or a shared object, the value is the virtual address of the storage unit affected by the relocation.

`r_info` This member gives both the symbol table index with respect to which the relocation must be made, and the type of relocation to apply. For example, a call instruction's relocation entry would hold the symbol table index of the function being called. If the index is `STN_UNDEF`, the undefined symbol index, the relocation uses 0 as the "symbol value." Relocation types are processor-specific; descriptions of their behavior appear below. When the text below refers to a relocation entry's relocation type or symbol table index, it means the result of applying `ELF32_R_TYPE` or `ELF32_R_SYM`, respectively, to the entry's `r_info` member.

```
#define ELF32_R_SYM(i)    ((i)>>8)
#define ELF32_R_TYPE(i) \
        ((unsigned char)(i))
#define ELF32_R_INFO(s,t) \
        (((s)<<8)+(unsigned char)(t))
```

`r_addend` This member specifies a constant addend used to compute the value to be stored into the relocatable field.

As shown above, only `Elf32_Rela` entries contain an explicit addend. Entries of type `Elf32_Rel` store an implicit addend in the location to be modified. Depending on the processor architecture, one form or the other might be necessary or more convenient. Con-

sequently, an implementation for a particular machine may use one form exclusively or either form depending on context.

A relocation section references two other sections: a symbol table and a section to modify. The section header's `sh_info` and `sh_link` members, described in "Section Header" on page 22-9, specify these relationships. Relocation entries for different object files have slightly different interpretations for the `r_offset` member.

- In relocatable files, `r_offset` holds a section offset. That is, the relocation section itself describes how to modify another section in the file; relocation offsets designate a storage unit within the second section.
- In executable and shared object files, `r_offset` holds a virtual address. To make these files' relocation entries more useful for the dynamic linker, the section offset (file interpretation) gives way to a virtual address (memory interpretation).

Although the interpretation of `r_offset` changes for different object files to allow efficient access by the relevant programs, the relocation types' meanings stay the same.

Relocation Types

Relocation entries describe how to alter the following instruction and data fields (bit numbers appear in the lower box corners; byte numbers appear in the upper box corners).

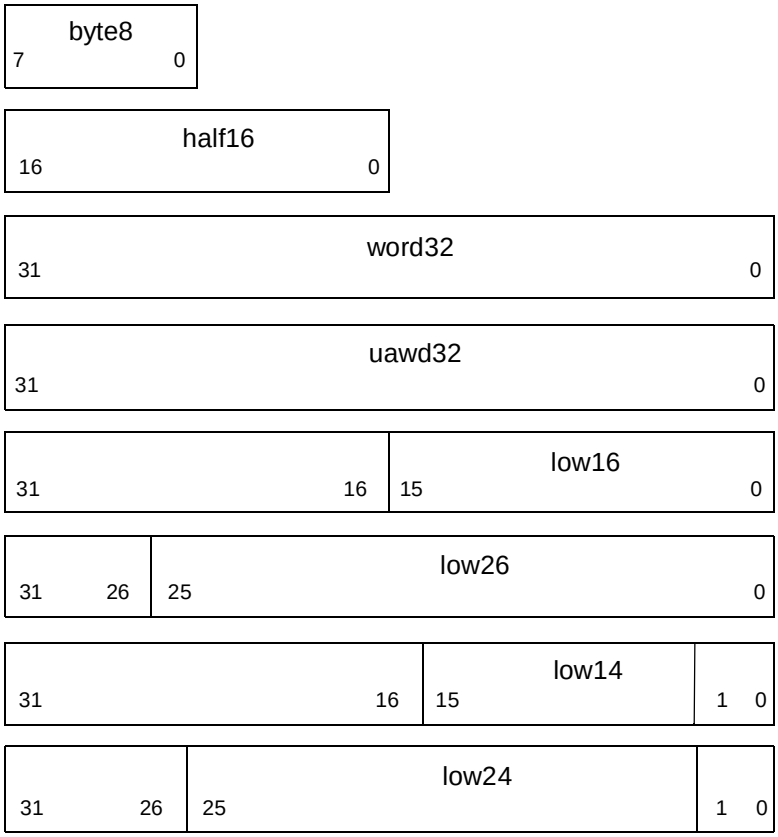
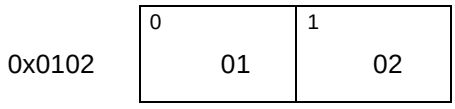


Figure 22-3. Relocatable Fields

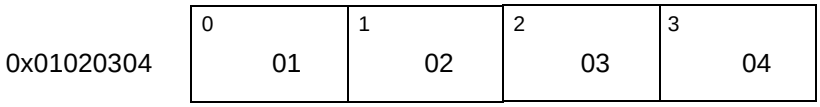
byte8 This specifies an 8-bit field occupying 1 byte with arbitrary alignment.

half16 This specifies a 16-bit field occupying 2 bytes with 2-byte alignment.



word32 This specifies a 32-bit field occupying 4 bytes with 4-byte alignment. These values use the byte order illustrated below.

uawd32 This specifies a 32-bit field occupying 4 bytes with arbitrary alignment. These values use the same byte order as for *word32*.



low16 This specifies a 16-bit field occupying the least significant bits of a field similar to *word32*. These bits represent values in the same byte order as *word32*.

low26 This specifies a 26-bit field occupying the least significant bits of a field similar to *word32*. These bits represent values in the same byte order as *word32*.

low14 This specifies a 14-bit field occupying the least significant bits (except for bits 1 and 0) of a field similar to *word32*. These bits represent values in the same byte order as *word32*.

low24 This specifies a 24-bit field occupying the least significant bits (except for bits 1 and 0) of a field similar to *word32*. These bits represent values in the same byte order as *word32*.

Calculations below assume the actions are transforming a relocatable file into either an executable or a shared object file. Conceptually, the link editor merges one or more relocatable files to form the output. It first decides how to combine and locate the input files, then updates the symbol values, and finally performs the relocation. Relocations applied to executable or shared object files are similar and accomplish the same result. Descriptions below use the following notation.

A This means the addend used to compute the value of the relocatable field.

AB This means the addressing base for the object.

B This means the base address at which a shared object has been loaded into memory during execution. The base address for an executable file is 0. Generally, a shared object file is built with a 0 base virtual address, but the execution address will be different. See “Program Header” on page 22-35 for more information about the base address.

G This means the place (section offset or address) of a global offset table entry for the symbol. See “Global Offset Table” on page 22-54 for more information.

GP This means the place (section offset or address) of a global offset table procedure entry for the symbol. See “Global Offset Table” on page 22-54 for more information.

L This means the place (section offset or address) of the procedure linkage table entry for a symbol. A procedure linkage table entry redirects a function call to the proper destination. The link editor builds the initial procedure linkage table, and the dynamic linker modifies the entries during execution. See “Procedure Linkage Table” on page 22-58 for more information.

P This means the place (section offset or address) of the storage unit being relocated (computed using `r_offset).`

S This means the value of the symbol whose index resides in the relocation entry.

Relocation entries apply to bytes (*byte8*), halfwords (*half16*), or words (the others). A relocation entry's `r_offset` value designates the offset or virtual address of the first byte of the affected storage unit. The relocation type specifies which bits to change and how to calculate their values. Because PowerUX uses only `Elf32_Rela` relocation entries with explicit addends, the `r_added` member serves as the relocation addend. In all cases, the addend and the computed result use the same byte order.

The following general rules apply to the interpretation of the relocation types shown below.

- "`+`" and "`-`" denote 32-bit modulus addition and subtraction, respectively. "`>>`" denotes arithmetic right shifting of the value of the left operand by the number of bits given by the right operand.
- For relocation types whose names end in "`_DISP14`" or "`_DISP16`", the upper 15 bits of the value computed before shifting must all be the same. For relocation types whose names end in "`_DISP24`" or "`_DISP26`", the upper 5 bits of the value computed before shifting must all be the same. For relocation types whose names end in either "`_DISP16`" or "`_DISP26`", the low 2 bits of the value computed before shifting must all be zero.
- A relocation type whose name ends in "`_DISP14`", "`_DISP16`", "`_DISP24`", or "`_DISP26`" must be used only in an instruction context, that is, where the target address computed from the relocated field is used as the destination of a transfer of control.
- For relocation types whose names end in "`_8`", the upper 24 bits of the computed value must all be zero. For relocation types whose names end in "`_8S`", the upper 25 bits of the computed value must all be the same. For relocation types whose names end in "`_14`" or "`_16`", the upper 16 bits of the computed value must all be zero. For relocation types whose names end in "`_16S`", the upper 17 bits of the computed value must all be the same.
- `uhi16(`*value*`)`, `hi16(`*value*`)` and `lo16(`*value*`)` denote the high, high, and low 16 bits, respectively, of the indicated value. The difference between `uhi16()` and `hi16()` is explained below.
- Reference in a calculation to the value "`G`" implicitly creates a global offset table entry for the indicated symbol. Reference in a calculation to the value "`GP`" implicitly creates a global offset table procedure entry for the indicated symbol. Reference in a calculation to the value "`L`" may implicitly create a procedure linkage table entry for the indicated symbol.
- For relocation types whose names begin with "`R_PPC_ABDIFF_`", "`R_PPC_ABREL_`", or "`R_PPC_SREL_`", the address represented by the symbol's value and the address of the storage unit affected by the relocation must both be in the same shared object, or both must be in an executable file.
- Where a relocation type does not use the associated symbol, the symbol index in the relocation entry must be zero.

- The link editor detects and reports violations of restrictions described above.

Table 22-24. Relocation Types

Name	Value	Field	Calculation
R_PPC_NONE	0	none	none
R_PPC_COPY	1	none	see below
R_PPC_GOTP_ENT	2	word32	see below
R_PPC_8	4	byte8	S + A
R_PPC_8S	5	byte8	S + A
R_PPC_16S	7	half16	S + A
R_PPC_14	8	low14	S + A
R_PPC_DISP16	8	low16	(S + A - P) >> 2
R_PPC_DISP14	9	low14	(S + A - P) >> 2
R_PPC_24	10	low24	S + A
R_PPC_DISP24	11	low24	(S + A - P) >> 2
R_PPC_PLT_DISP24	14	low24	(L + A - P) >> 2
R_PPC_BBASED_16HU	15	half16	uhi16(B + A)
R_PPC_BBASED_32	16	word32	B + A
R_PPC_BBASED_32UA	17	uawd32	B + A
R_PPC_BBASED_16H	18	half16	hi16(B + A)
R_PPC_BBASED_16L	19	half16	lo16(B + A)
R_PPC_ABDIFF_16HU	23	half16	uhi16(AB - S + A)
R_PPC_ABDIFF_32	24	word32	AB - S + A
R_PPC_ABDIFF_32UA	25	uawd32	AB - S + A
R_PPC_ABDIFF_16H	26	half16	hi16(AB - S + A)
R_PPC_ABDIFF_16L	27	half16	lo16(AB - S + A)
R_PPC_ABDIFF_16	28	half16	AB - S + A
R_PPC_16HU	31	half16	uhi16(S + A)
R_PPC_32	32	word32	S + A
R_PPC_32UA	33	uawd32	S + A
R_PPC_16H	34	half16	hi16(S + A)
R_PPC_16L	35	half16	lo16(S + A)
R_PPC_16	36	half16	S + A
R_PPC_GOT_16HU	39	half16	uhi16(G + A)
R_PPC_GOT_32	40	word32	G + A

Table 22-24. Relocation Types (Cont.)

Name	Value	Field	Calculation
R_PPC_GOT_32UA	41	uawd32	G + A
R_PPC_GOT_16H	42	half16	hi16(G + A)
R_PPC_GOT_16L	43	half16	lo16(G + A)
R_PPC_GOT_16	44	half16	G + A
R_PPC_GOTP_16HU	47	half16	uhi16(GP + A)
R_PPC_GOTP_32	48	word32	GP + A
R_PPC_GOTP_32UA	49	uawd32	GP + A
R_PPC_GOTP_16H	50	half16	hi16(GP + A)
R_PPC_GOTP_16L	51	half16	lo16(GP + A)
R_PPC_GOTP_16	52	half16	GP + A
R_PPC_PLT_16HU	55	half16	uhi16(L + A)
R_PPC_PLT_32	56	word32	L + A
R_PPC_PLT_32UA	57	uawd32	L + A
R_PPC_PLT_16H	58	half16	hi16(L + A)
R_PPC_PLT_16L	59	half16	lo16(L + A)
R_PPC_PLT_16	60	half16	L + A
R_PPC_ABREL_16HU	63	half16	uhi16(S + A - AB) (See text below)
R_PPC_ABREL_32	64	word32	S + A - AB (See text below)
R_PPC_ABREL_32UA	65	uawd32	S + A - AB (See text below)
R_PPC_ABREL_16H	66	half16	hi16(S + A - AB) (See text below)
R_PPC_ABREL_16L	67	half16	lo16(S + A - AB) (See text below)
R_PPC_ABREL_16	68	half16	S + A - AB (See text below)
R_PPC_GOT_ABREL_16HU	71	half16	uhi16(G + A - AB)
R_PPC_GOT_ABREL_32	72	word32	G + A - AB
R_PPC_GOT_ABREL_32UA	73	uawd32	G + A - AB
R_PPC_GOT_ABREL_16H	74	half16	hi16(G + A - AB)
R_PPC_GOT_ABREL_16L	75	half16	lo16(G + A - AB)
R_PPC_GOT_ABREL_16	76	half16	G + A - AB
R_PPC_GOTP_ABREL_16HU	79	half16	uhi16(GP + A - AB)

Table 22-24. Relocation Types (Cont.)

Name	Value	Field	Calculation
R_PPC_GOTP_ABREL_32	80	word32	GP + A - AB
R_PPC_GOTP_ABREL_32UA	81	uawd32	GP + A - AB
R_PPC_GOTP_ABREL_16H	82	half16	hi16(GP + A - AB)
R_PPC_GOTP_ABREL_16L	83	half16	lo16(GP + A - AB)
R_PPC_GOTP_ABREL_16	84	half16	GP + A - AB
R_PPC_PLT_ABREL_16HU	87	half16	uhi16(L + A - AB)
R_PPC_PLT_ABREL_32	88	word32	L + A - AB
R_PPC_PLT_ABREL_32UA	89	uawd32	L + A - AB
R_PPC_PLT_ABREL_16H	90	half16	hi16(L + A - AB)
R_PPC_PLT_ABREL_16L	91	half16	lo16(L + A - AB)
R_PPC_PLT_ABREL_16	92	half16	L + A - AB
R_PPC_SREL_16HU	95	half16	uhi16(S + A - P)
R_PPC_SREL_32	96	word32	S + A - P
R_PPC_SREL_32UA	97	uawd32	S + A - P
R_PPC_SREL_16H	98	half16	hi16(S + A - P)
R_PPC_SREL_16L	99	half16	lo16(S + A - P)
R_PPC_REL_EXT_1	254	word32	See text below
R_PPC_REL_EXT_2	255	word32	See text below

The semantics of `hi16()` are different from those of u`hi16()`. For `hi16()`, if bit 16 of the 32-bit operand value is set, then a value of 1 is added to the high-order 16 bits of the 32-bit operand.

Some relocation types have semantics beyond simple calculation.

`R_PPC_COPY` This relocation type assists dynamic linking. Its offset member refers to a location in a writable segment. The symbol table index specifies a symbol that should exist both in the current object file and in a shared object. During execution, the dynamic linker copies data associated with the shared object's symbol to the location specified by the object.

`R_PPC_GOTP_ENT` This relocation type assists dynamic linking. The relocation offset gives the location of a global offset table procedure entry. The relocation symbol names the procedure. The relocation addend gives the address of the associated GOTP binding entry. For an executable file, this address is absolute; for a shared object file, it is relative to the base address for the shared object. The use of relocation types whose names end in "`_16`" is generally subject to failure, because the value computed may not fit in 16 bits. However, the use of the `R_PPC_GOT_ABREL_16` and

R_PPC_GOTP_ABREL_16 relocation types does not fail unless the total number of distinct GOT and GOTP entries for the executable or shared object being link edited exceeds 16,380. In other words, the link editor is obliged to favor GOT and GOTP entries when choosing an addressing base and laying out the private data of either the executable or shared object file.

R_PPC_GOT_ABREL_16 and R_PPC_GOTP_ABREL_16 relocation types do not fail unless the total number of distinct GOT and GOTP entries for the executable or shared object being link edited exceeds 16,380. In other words, the link editor is obliged to favor GOT and GOTP entries when choosing an addressing base and laying out the private data of either the executable or shared object file.

The relocation types that typically remain after link editing and which require processing by the dynamic linker include R_PPC_COPY, R_PPC_GOTP_ENT, the R_PPC_BBASED family, and R_PPC_32. However, the dynamic linker is prepared to handle all relocation types except those whose calculations involve any of the values "G", "GP", and "L".

R_PPC_REL_EXT_1 The PowerUX implementation of DWARF symbolic debugging information requires an ability to subtract two symbolic definitions to obtain a single symbolic reference. This relocation type is equivalent to R_PPC_32, for the minuend. The subtrahend is represented in the next relocation entry.

R_PPC_REL_EXT_2 This relocation type identifies a relocation entry which must appear immediately after the corresponding relocation entry of type R_PPC_REL_EXT_1. This relocation entry pertains to the subtrahend.

The R_PPC_ABREL_* relocation types have a different calculation when the **ld(1) -QAda option** is used. In this case, the base address of the shared object, rather than the addressing base, is used in the calculations.

Program Execution

This section describes the object file information and system actions that create running programs.

Executable and shared object files statically represent programs. To execute such programs, the system uses the files to create dynamic program representations, or process images.

Program Header

An executable or shared object file's program header table is an array of structures, each describing a segment or other information the system needs to prepare the program for execution. An object file *segment* contains one or more *sections* as "Segment Contents" on page 22-40 describes.

Program headers are meaningful only for executable and shared object files. A file specifies its own program header size with the ELF header's `e_phentsize` and `e_phnum` members (see "ELF Header" on page 22-3).

```
typedef struct {
    Elf32_Word  p_type;
    Elf32_Off   p_offset;
    Elf32_Addr  p_vaddr;
    Elf32_Addr  p_paddr;
    Elf32_Word  p_filesz;
    Elf32_Word  p_memsz;
    Elf32_Word  p_flags;
    Elf32_Word  p_align;
} Elf32_Phdr;
```

`p_type` — This member tells what kind of segment this array element describes or how to interpret the array element's information. Type values and their meanings appear below.

`p_offset` — This member gives the offset from the beginning of the file at which the first byte of the segment resides.

`p_vaddr` — This member gives the virtual address at which the first byte of the segment resides in memory.

`p_paddr` — On systems for which physical addressing is relevant, this member is reserved for the segment's physical address. Because PowerUX ignores physical addressing for application programs, this member has unspecified contents for executable files and shared objects.

`p_filesz` — This member gives the number of bytes in the file image of the segment; it may be zero.

`p_memsz` — This member gives the number of bytes in the memory image of the segment; it may be zero.

`p_flags` — This member gives flags relevant to the segment. Defined flag values appear in Table 22-25.

`p_align` — As "Program Linking" on page 22-3 describes, loadable process segments must have congruent values for `p_vaddr` and `p_offset`, modulo the page size. This member gives the value to which the segments are aligned in memory and in the file. Values 0 and 1 mean no alignment is required. Otherwise, `p_align` should be a positive, integral power of 2, and `p_vaddr` should equal `p_offset`, modulo `p_align`.

Some entries describe process segments; others give supplementary information and do not contribute to the process image. Segment entries may appear in any order, except as

explicitly noted below. Defined type values follow; other values are reserved for future use.

Table 22-25. Segment Types, p_type

Name	Value
PT_NULL	0
PT_LOAD	1
PT_DYNAMIC	2
PT_INTERP	3
PT_NOTE	4
PT_SHLIB	5
PT_PHDR	6
PT_LOPROC	0x70000000
PT_PPC_DEBINFADDR	0x70000001
PT_PPC_VENDOR	0x7fffffff
PT_HIPROC	0x7fffffff

PT_NULL — The array element is unused; other members' values are undefined. This type lets the program header table have ignored entries.

PT_LOAD — The array element specifies a loadable segment, described by p_filesz and p_memsz. The bytes from the file are mapped to the beginning of the memory segment. If the segment's memory size (p_memsz) is larger than the file size (p_filesz), the "extra" bytes are defined to hold the value 0 and to follow the segment's initialized area. The file size may not be larger than the memory size. Loadable segment entries in the program header table appear in ascending order, sorted on the p_vaddr member.

PT_DYNAMIC — The array element specifies dynamic linking information. See "Dynamic Section" on page 22-47 for more information.

PT_INTERP — The array element specifies the location and size of a null-terminated path name to invoke as an interpreter. This segment type is meaningful only for executable files (though it may occur for shared objects); it may not occur more than once in a file. If it is present, it must precede any loadable segment entry. See "Program Interpreter" on page 22-45 for further information.

PT_NOTE — The array element specifies the location and size of auxiliary information. See "Note Section" on page 22-41 for details.

PT_SHLIB — This segment type is reserved but has unspecified semantics.

`PT_PHDR` The array element, if present, specifies the location and size of the program header table itself, both in the file and in the memory image of the program. This segment type may not occur more than once in a file. Moreover, it may occur only if the program header table is part of the memory image of the program. If it is present, it must precede any loadable segment entry. See "Program Interpreter" on page 22-45 for further information.

`PT_LOPROC` through `PT_HIPROC`
Values in this inclusive range are reserved for processor-specific semantics.

`PT_PPC_DEBINFADDR`
The array element, if present, specifies the address of the "text description" (tdesc) *debug info* protocol.

`PT_PPC_VENDOR` The array element, if present, specifies the address of the vendor section.

Unless specifically required elsewhere, all program header segment types are optional. That is, a file's program header table may contain only those elements relevant to its contents.

No two loadable segments in an executable or shared object file occupy the same 64K region. More precisely, given the virtual addresses of any two bytes in different loadable segments of an executable or shared object file, the integer quotients of those addresses, upon division by 64, differ. Executable and writable sections must occupy different 256M regions.

For every loadable segment in an executable or shared object file, if that segment does not have write permission, then either the segment's `p_filesz` value is zero, or the segment's `p_memsz` and `p_filesz` values are the same.

No segment defined in an executable file occupies space at or above address 0x80000000.

An executable file defines at least one writable segment, that is, a segment with write permission.

The *break area* is a writable area of memory whose size can be increased by the application. (See **`brk(2)`**.) The *break* value defines the current extent of the break area. The initial break value is the end of the highest writable segment in the executable file. More precisely, the initial break value is the sum of the `p_vaddr` and `p_memsz` values for the executable file's writable segment with largest `p_vaddr` value. The break value is not set lower than its initial value.

If the program header in a shared object file contains a `PT_INTERP` array element, then it also contains a `PT_PHDR` array element.

Base Address

Executable and shared object files have a *base address* which is the lowest virtual address associated with the memory image of the program's object file. One use of the base address is to relocate the memory image of the program during dynamic linking. An executable or shared object file's base address is calculated during execution from three val-

ues: the memory load address, the maximum page size, and the lowest virtual address of a program's loadable segment. As "Program Loading" on page 22-42 describes, the virtual addresses in the program headers might not represent the actual virtual addresses of the program's memory image. To compute the base address, one determines the memory address associated with the lowest `p_vaddr` value for a `PT_LOAD` segment. One then obtains the base address by truncating the memory address to the nearest multiple of the maximum page size. Depending on the kind of file being loaded into memory, the memory address might or might not match the `p_vaddr` values.

Segment Permissions

A program to be loaded by the system must have at least one loadable segment (although this is not required by the file format). When the system creates loadable segments' memory images, it gives access permissions as specified in the `p_flags` member. All bits included in the `PF_MASKPROC` mask are reserved for processor-specific semantics.

Table 22-26. Segment Flag Bits, p_flags

Name	Value	Meaning
`PF_X`	0x1	Execute
`PF_W`	0x2	Write
`PF_R`	0x4	Read
`PF_MASKPROC`	0xf0000000	Unspecified

If a permission bit is 0, that type of access is denied. Actual memory permissions depend on the memory management unit, which may vary from one system to another. Although all flag combinations are valid, the system may grant more access than requested. In no case, however, will a segment have write permission unless it is specified explicitly. The following table shows both the exact flag interpretation and the allowable flag interpretation.

Table 22-27. Segment Permissions

Flags	Value	Exact	Allowable
none	0	All access denied	All access denied
`PF_X`	1	Execute only	Read, execute
`PF_W`	2	Write only	Read, write, execute
`PF_W + PF_X`	3	Write, execute	Read, write, execute
`PF_R`	4	Read only	Read, execute

Table 22-27. Segment Permissions

Flags	Value	Exact	Allowable
`PF_R + PF_X`	5	Read, execute	Read, execute
`PF_R + PF_W`	6	Read, write	Read, write, execute
`PF_R + PF_W + PF_X`	7	Read, write, execute	Read, write, execute

For example, typical text segments have read and execute - but not write - permissions. Data segments normally have read, write, and execute permissions.

Segment Contents

An object file segment comprises one or more sections, though this fact is transparent to the program header. Whether the file segment holds one or many sections also is immaterial to program loading. Nonetheless, various data must be present for program execution, dynamic linking, and so on. The diagrams below illustrate segment contents in general terms. The order and membership of sections within a segment may vary; moreover, processor-specific constraints may alter the examples below.

Text segments contain read-only instructions and data, typically including the following sections described earlier in this chapter. Other sections may also reside in loadable segments; these examples are not meant to give complete and exclusive segment contents.

Table 22-28. Text Segment

`.text`
`.rodata`
`.hash`
`.dynsym`
`.dynstr`
`.plt`
`.rel.got`

Data segments contain writable data and instructions, typically including the following sections.

Table 22-29. Data Segment

`.data`

Table 22-29. Data Segment

.dynamic
.got
.bss

A PT_DYNAMIC program header element points at the **.dynamic** section, explained in "Dynamic Section" on page 22-47. The **.got** and **.plt** sections also hold information related to position-independent code and dynamic linking. Although the **.plt** appears in a text segment above, it may reside in a text or a data segment, depending on the processor. See "Global Offset Table" on page 22-54 and "Procedure Linkage Table" on page 22-58 for details.

As described in "Section Header" on page 22-9, the **.bss** section has the type SHT_NOBITS. Although it occupies no space in the file, it contributes to the segment's memory image. Normally, these uninitialized data reside at the end of the segment, thereby making p_memsz larger than p_filesz in the associated program header element.

Note Section

Sometimes a vendor or system builder needs to mark an object file with special information that other programs will check for conformance, compatibility, and so forth. Sections of type SHT_NOTE and program header elements of type PT_NOTE can be used for this purpose. The note information in sections and program header elements holds any number of entries, each of which is an array of 4-byte words in the format of the target processor. Labels in Table 22-30 help explain note information organization, but they are not part of the specification.

Table 22-30. Note Information

namesz
descsz
type
name ... desc ...

namesz and name — The first namesz bytes in name contain a null-terminated character representation of the entry's owner or originator. There is no formal mechanism for avoiding name conflicts. By convention, vendors use their own name, such as "XYZ Computer Company," as the identifier. If no name is present, namesz contains 0. Padding is present, if necessary, to ensure 4-byte alignment for the descriptor. Such padding is not included in namesz.

`descsz` and `desc` — The first `descsz` bytes in `desc` hold the note descriptor. If no descriptor is present, `descsz` contains 0. Padding is present, if necessary, to ensure 4-byte alignment for the next note entry. Such padding is not included in `descsz`.

`type` — This word gives the interpretation of the descriptor. Each originator controls its own types; multiple interpretations of a single type value may exist. Thus, a program must recognize both the name and the type to "understand" a descriptor. Types currently must be non-negative.

To illustrate, the following note segment holds two entries.

Table 22-31. Example Note Segment

	+ 0	+ 1	+ 2	+ 3	
namesz	7				
descsz	0				No descriptor
type	1				
name	X	Y	Z		
	C	o	\0	*pad*	
namesz	7				
descsz	8				
type	3				
name	X	Y	Z		
	C	o	\0	*pad*	
desc	*word 0*				
	word 1				

The system reserves note information with no name (`namesz==0`) and with a zero-length name (`name[0]=='\0'`) but currently defines no types. All other names must have at least one non-null character.

Program Loading

As the system creates or augments a process image, it logically copies a file's segment to a virtual memory segment. When--and if--the system physically reads the file depends on the program's execution behavior, system load, and so forth. A process does not require a physical page unless it references the logical page during execution, and processes commonly leave many pages unreferenced. Therefore, delaying physical reads frequently obviates them, improving system performance. To obtain this efficiency in practice, executable and shared object files must have segment images whose file offsets and virtual addresses are congruent, modulo the page size.

The virtual address (`p_vaddr`) and file offset (`p_vaddr`) for segments are congruent, modulo 64K (`0x10000`). The value of the `p_align` member of each program header element in an executable or shared object file is 64K. The following examples show 64K alignment.

Table 22-32. Executable File

File Offset	File	Virtual Address
0	ELF Header	
	Program Header Table	
	Other Information	
0x1000	Text Segment ... size = 0xaf48 bytes	0x10001000
0xc000	RO Data Segment ... size = 0x430 bytes	0x1003c000
0xd000	Data Segment ... size = 0x113c bytes	0x3000d000
0xe13c	Other Information ...	

Table 22-33. Program Header Segments

Member	Text	Data
`p_type`	`PT_LOAD`	`PT_LOAD`
`p_offset`	0x100	0x2bf00
`p_vaddr`	0x10100	0x4bf00
`p_paddr`	unspecified	unspecified
`p_filesz`	0x2be00	0x4e00
`p_memsz`	0x2be00	0x5e24
`p_flags`	`PF_R + PF_X`	`PF_R + PF_W`
`p_align`	0x10000	0x10000

Although the example's file offsets and virtual addresses are congruent modulo 64K for both text and data, up to four file pages hold impure text or data (depending on page size and file system block size).

- The first text page contains the ELF header, the program header table, and other information.
- The last text page holds a copy of the beginning of data.
- The first data page has a copy of the end of text.
- The last data page may contain file information not relevant to the running process.

Logically, the system enforces the memory permissions as if each segment were complete and separate; segments' addresses are adjusted to ensure each logical page in the address space has a single set of permissions. In the example above, the region of the file holding the end of text and the beginning of data will be mapped twice: at one virtual address for text and at a different virtual address for data.

The end of the data segment requires special handling for uninitialized data, which the system defines to begin with zero values. Thus if a file's last data page includes information not in the logical memory page, the extraneous data must be set to zero, not the unknown contents of the executable file. "Impurities" in the other three pages are not logically part of the process image; whether the system expunges them is unspecified. The memory image for this program follows, assuming 4 KB (`0x1000`) pages.

Table 22-34. Process Image Segments

Virtual Address	Contents
0x20000	*Header Padding* 0xe0 zero bytes
0x200e0	Text Segment . . . 0x6458 bytes
0x26538	*RO Data Padding* `0xac8` zero bytes
0x46000	*Text Padding* 0x538 zero bytes
0x46538	RO Data Segment . . . 0x18 bytes
0x46550	*Data Padding* 0xab0 zero bytes
0x56000	*RO Data Padding* 0x550 zero bytes
0x56550	Data Segment . . . 0x5884 bytes

Table 22-34. Process Image Segments (Cont.)

Virtual Address	Contents
0x574c0	Uninitialized Data `0x4914` zero bytes
0x5bdd4	*Page Padding* 0x230 zero bytes

Hardware requires that pages be “pure”; thus sections always start on a page boundary.

One aspect of segment loading differs between executable files and shared objects. Executable file segments typically contain absolute code. To let the process execute correctly, the segments must reside at the virtual addresses used to build the executable file. Thus the system uses the `p_vaddr` values unchanged as virtual addresses.

On the other hand, shared object segments typically contain position-independent code. (For background, see Chapter 4 (“Link Editor and Linking”).) This lets a segment’s virtual address change from one process to another, without invalidating execution behavior. Though the system chooses virtual addresses for individual processes, it maintains the segments’ *relative positions*. Because position-independent code uses relative addressing between segments, the difference between virtual addresses in memory must match the difference between virtual addresses in the file. The following table shows possible shared object virtual address assignments for several processes, illustrating constant relative positioning. The table also illustrates the base address computations.

Table 22-35. Example Shared Object Segment Addresses

Source	Text	Data	Base Address
File	0x200	0x2a400	0x0
Process 1	0xc0000200	0xc002a400	0xc0000000
Process 2	0xc0010200	0xc003a400	0xc0010000
Process 3	0xd0020200	0xd004a400	0xd0020000
Process 4	0xd0030200	0xd005a400	0xd0030000

Program Interpreter

An executable file may have one `PT_INTERP` program header element. During `exec()`, the system retrieves a path name from the `PT_INTERP` segment and creates the initial process image from the interpreter file’s segments. That is, instead of using the original executable file’s segment images, the system composes a memory image for the interpreter. It then is the interpreter’s responsibility to receive control from the system and provide an environment for the application program.

The interpreter receives control in one of two ways. First, it may receive a file descriptor to read the executable file, positioned at the beginning. It can use this file descriptor to read and/or map the executable file's segments into memory. Second, depending on the executable file format, the system may load the executable file into memory instead of giving the interpreter an open file descriptor. With the possible exception of the file descriptor, the interpreter's initial process state matches what the executable file would have received. The interpreter itself may not require a second interpreter. An interpreter may be either a shared object or an executable file.

- A shared object (the normal case) is loaded as position-independent, with addresses that may vary from one process to another; the system creates its segments in the dynamic segment area used by `mmap()` and related services. Consequently, a shared object interpreter typically will not conflict with the original executable file's original segment addresses.
- An executable file is loaded at fixed addresses; the system creates its segments using the virtual addresses from the program header table. Consequently, an executable file interpreter's virtual addresses may collide with the first executable file; the interpreter is responsible for resolving conflicts.
- The default program interpreter on PowerUX is **`/usr/lib/libc.so.1`**.

Dynamic Linker

When building an executable file that uses dynamic linking, the link editor adds a program header element of type `PT_INTERP` to an executable file, telling the system to invoke the dynamic linker as the program interpreter. `exec()` and the dynamic linker cooperate to create the process image for the program, which entails the following actions:

- Adding the executable file's memory segments to the process image;
- Adding shared object memory segments to the process image;
- Performing relocations for the executable file and its shared objects;
- Closing the file descriptor that was used to read the executable file, if one was given to the dynamic linker;
- Transferring control to the program, making it look as if the program had received control directly from `exec()`.

The link editor also constructs various data that assist the dynamic linker for executable and shared object files. As shown in "Program Header" on page 22-35, these data reside in loadable segments, making them available during execution.

- A **`.dynamic`** section with type `SHT_DYNAMIC` holds various data. The structure residing at the beginning of the section holds the addresses of other dynamic linking information.
- The **`.hash`** section with type `SHT_HASH` holds a symbol hash table.
- The **`.got`** and **`.plt`** sections with type `SHT_PROGBITS` hold two separate tables: the global offset table and the procedure linkage table. Sections

below explain how the dynamic linker uses and changes the tables to create memory images for object files.

As explained in "Program Loading" on page 22-42, shared objects may occupy virtual memory addresses that are different from the addresses recorded in the file's program header table. The dynamic linker relocates the memory image, updating absolute addresses before the application gains control. Although the absolute address values would be correct if the library were loaded at the addresses specified in the program header table, this normally is not the case.

If the process environment contains a variable named `LD_BIND_NOW` with a non-null value, the dynamic linker processes all relocations before transferring control to the program. For example, all the following environment entries would specify this behavior.

- `LD_BIND_NOW=1`
- `LD_BIND_NOW=on`
- `LD_BIND_NOW=off`

Otherwise, `LD_BIND_NOW` either does not occur in the environment or has a null value. In this case, dynamic linker is permitted to evaluate procedure linkage table entries lazily, thus avoiding symbol resolution and relocation overhead for functions that are not called. See "Procedure Linkage Table" on page 22-58 for more information.

Dynamic Section

If an object file participates in dynamic linking, its program header table will have an element of type `PT_DYNAMIC`. This "segment" contains the **`.dynamic`** section. A special symbol, `_DYNAMIC`, labels the section, which contains an array of the following structures.

```
typedef struct {
    Elf32_Sword     d_tag;
    union {
        Elf32_Word  d_val;
        Elf32_Addr  d_ptr;
    } d_un;
} Elf32_Dyn;

extern Elf32_Dyn    _DYNAMIC[];
```

For each object with this type, `d_tag` controls the interpretation of `d_un`.

`d_val` These `Elf32_Word` objects represent integer values with various interpretations.

`d_ptr` These `Elf32_Addr` objects represent program virtual addresses. As mentioned previously, a file's virtual addresses might not match the memory virtual addresses during execution. When interpreting addresses contained in the dynamic structure, the dynamic linker computes actual addresses, based on the original file value and the memory base address. For consistency, files do not contain relocation entries to "correct" addresses in the dynamic structure.

The following table summarizes the tag requirements for executable and shared object files. If a tag is marked “mandatory,” then the dynamic linking array must have an entry of that type. Likewise, “optional” means an entry for the tag may appear but is not required.

Table 22-36. Dynamic Array Tags, d_tag

Name	Value	d_un	Executable	Shared Object
DT_NULL	0	ignored	mandatory	mandatory
DT_NEEDED	1	d_val	optional	optional
DT_PLTRELSZ	2	d_val	optional	optional
DT_PLTGOT	3	d_ptr	optional	optional
DT_HASH	4	d_ptr	mandatory	mandatory
DT_STRTAB	5	d_ptr	mandatory	mandatory
DT_SYMTAB	6	d_ptr	mandatory	mandatory
DT_RELA	7	d_ptr	mandatory	optional
DT_RELASZ	8	d_val	mandatory	optional
DT_RELAENT	9	d_val	mandatory	optional
DT_STRSZ	10	d_val	mandatory	mandatory
DT_SYMENT	11	d_val	mandatory	mandatory
DT_INIT	12	d_ptr	optional	optional
DT_FINI	13	d_ptr	optional	optional
DT_SONAME	14	d_val	ignored	optional
DT_RPATH	15	d_val	optional	ignored
DT_SYMBOLIC	16	ignored	ignored	optional
DT_REL	17	d_ptr	mandatory	optional
DT_RELSZ	18	d_val	mandatory	optional
DT_RELENT	19	d_val	mandatory	optional
DT_PLTREL	20	d_val	optional	optional
DT_DEBUG	21	d_ptr	optional	ignored
DT_TEXTREL	22	ignored	optional	optional
DT_JMPREL	23	d_ptr	optional	optional
DT_LOPROC	0x70000000	unspecified	unspecified	unspecified
DT_PPC_ADDRBASE	0x70000001	d_ptr	optional	required
DT_PPC_PLTSTART	0x70000002	d_ptr	optional	optional
DT_PPC_PLTEND	0x70000003	d_ptr	optional	optional
DT_PPC_TDESC	0x70000004	d_ptr	optional	optional

Table 22-36. Dynamic Array Tags, d_tag (Cont.)

Name	Value	d_un	Executable	Shared Object
DT_PPC_ARMS	0x70000100	d_val	optional	optional
DT_PPC_BIND_SYM	0x70000101	d_ptr	optional	optional
DT_HIPROC	0x7fffffff	unspecified	unspecified	unspecified

DT_NULL — An entry with a DT_NULL tag marks the end of the _DYNAMIC array.

DT_NEEDED — This element holds the string table offset of a null-terminated string, giving the name of a needed library. The offset is an index into the table recorded in the DT_STRTAB entry. See "Shared Object Dependencies" on page 22-52 for more information about these names. The dynamic array may contain multiple entries with this type. These entries' relative order is significant, though their relation to entries of other types is not.

DT_PLTRELSZ — This element holds the total size, in bytes, of the relocation entries associated with the global offset table. This relocation table contains all relocation entries of type R_PPC_GOTP_ENT, and only those entries. In particular, relocation entries applying to the procedure linkage table are found with all other relocation entries in the relocation table specified by the DT_RELA, DT_RELASZ, and DT_RELAENT entries.

DT_PLTGOT — This element holds an address of three consecutive words in the private data of an executable or shared object file. These 12 bytes are 4-byte aligned. The first word is set by the link editor and contains the address of the symbol _DYNAMIC; the address is absolute for an executable file and relative to the base address for a shared object. The second word is set by the dynamic linker and points to the link map entry for the object (see below). The third word is used to support lazy binding. The DT_PLTGOT entry is required in every object file that participates in dynamic linking. The link editor chooses where to locate the three words, usually at the beginning of the global offset table.

DT_HASH — This element holds the address of the symbol hash table, described in "Hash Table" on page 22-59.

DT_STRTAB — This element holds the address of the string table, described in the first part of this chapter. Symbol names, library names, and other strings reside in this table.

DT_SYMTAB — This element holds the address of the symbol table, described in the first part of this chapter, with Elf32_Sym entries for the 32-bit class of files.

DT_RELA — This element holds the address of a relocation table, described in the first part of this chapter. Entries in the table have explicit addends, such as Elf32_Rela for the 32-bit file class. An object

file may have multiple relocation sections. When building the relocation table for an executable or shared object file, the link editor concatenates those sections to form a single table. Although the sections remain independent in the object file, the dynamic linker sees a single table. When the dynamic linker creates the process image for an executable file or adds a shared object to the process image, it reads the relocation table and performs the associated actions. If this element is present, the dynamic structure must also have DT_RELASZ and DT_RELAENT elements. When relocation is "mandatory" for a file, either DT_RELA or DT_REL may occur (both are permitted but not required).

DT_RELASZ — This element holds the total size, in bytes, of the DT_RELA relocation table.

DT_RELAENT — This element holds the size, in bytes, of the DT_RELA relocation entry.

DT_STRSZ — This element holds the size, in bytes, of the string table.

DT_SYMENT — This element holds the size, in bytes, of a symbol table entry.

DT_INIT — This element holds the address of the initialization function, discussed in "Initialization and Termination Functions" on page 22-60.

DT_FINI — This element holds the address of the termination function, discussed in "Initialization and Termination Functions" on page 22-60.

DT_SONAME — This element holds the string table offset of a null-terminated string, giving the name of the shared object. The offset is an index into the table recorded in the DT_STRTAB entry. See "Shared Object Dependencies" on page 22-52 for more information about these names.

DT_RPATH — This element holds the string table offset of a null-terminated search library search path string, discussed in "Shared Object Dependencies" on page 22-52. The offset is an index into the table recorded in the DT_STRTAB entry.

DT_SYMBOLIC — This element's presence in a shared object library alters the dynamic linker's symbol resolution algorithm for references within the library. Instead of starting a symbol search with the executable file, the dynamic linker starts from the shared object itself. If the shared object fails to supply the referenced symbol, the dynamic linker then searches the executable file and other shared objects as usual.

DT_REL — This element is not used on PowerUX.

DT_RELSZ — This element is not used on PowerUX.

DT_RELENT — This element is not used on PowerUX.

DT_PLTREL — This member specifies the type of relocation entry to which the global offset table refers. This relocation table contains all relocation entries of type R_PPC_GOTP_ENT, and only those entries. In particular, relocation entries applying to the procedure linkage table are found with all other relocation entries in the relocation table specified by the DT_RELA, DT_RELASZ, and DT_RELAENT entries.

DT_DEBUG — This member is used for debugging.

DT_TEXTREL — This member's absence signifies that no relocation entry should cause a modification to a non-writable segment, as specified by the segment permissions in the program header table. If this member is present, one or more relocation entries might request modifications to a non-writable segment, and the dynamic linker can prepare accordingly.

DT_JMPREL — This element holds the total size, in bytes, of the relocation entries associated with the global offset table. This relocation table contains all relocation entries of type R_PPC_GOTP_ENT, and only those entries. In particular, relocation entries applying to the procedure linkage table are found with all other relocation entries in the relocation table specified by the DT_RELA, DT_RELASZ, and DT_RELAENT entries.

DT_LOPROC through DT_HIPROC — Values in this inclusive range are reserved for processor-specific semantics.

DT_PPC_ADDRBASE — This entry's d_ptr member gives the address base for the object file. If this entry is missing for an executable that participates in dynamic linking, the addressing base is 0.

DT_PPC_PLTSTART — This entry's d_ptr member gives the low address (inclusive) of the PLT region in an object file. If this entry is present, then DT_PPC_PLTEND is also present.

DT_PPC_PLTEND — This entry's d_ptr member gives the high address (exclusive) of the PLT region in an object file. If this entry is present, then DT_PPC_PLTSTART is also present.

DT_PPC_TDESC — This entry's d_ptr member gives the low address (inclusive) of the "text description" (tdesc) information in an object file.

Except for the DT_NULL element at the end of the array, and the relative order of DT_NEEDED elements, entries may appear in any order. Tag values not appearing in the table are reserved.

The *PLT* region is that portion of an object file that is made executable by the dynamic linker after relocations are performed in the region. The PLT region includes all PLT entries for the object file that require relocation by the dynamic linker. The region of memory between (((DT_PPC_PLTSTART value) / 64K) * 64K) (inclusive) and ((((DT_PPC_PLTEND value) + 64K - 1) / 64K) *

64K) (exclusive) is subject to being made executable by the dynamic linker.

`DT_PPC_ARMS` This entry is present if the object was link edited with a map file that defined a Concurrent Ada ARMS segment.

`DT_PPC_BIND_SYM` This entry is present if a function is to be non-lazily bound to the shared object during dynamic linking. One or more of these entries may be present in a shared object. The dynamic linker fully binds each named symbol to the shared object, even if other symbols are lazily bound.

Shared Object Dependencies

When the link editor processes an archive library, it extracts library members and copies them into the output object file. These statically linked services are available during execution without involving the dynamic linker. Shared objects also provide services, and the dynamic linker must attach the proper shared object files to the process image for execution. Thus executable and shared object files describe their specific dependencies.

When the dynamic linker creates the memory segments for an object file, the dependencies (recorded in `DT_NEEDED` entries of the dynamic structure) tell what shared objects are needed to supply the program's services. By repeatedly connecting referenced shared objects and their dependencies, the dynamic linker builds a complete process image. When resolving symbolic references, the dynamic linker examines the symbol tables with a breadth-first search. That is, it first looks at the symbol table of the executable program itself, then at the symbol tables of the `DT_NEEDED` entries (in order), then at the second level `DT_NEEDED` entries, and so on.

Even when a shared object is referenced multiple times in the dependency list, the dynamic linker will connect the object only once to the process.

Names in the dependency list are copies either of the `DT_SONAME` strings or the path names of the shared objects used to build the object file. For example, if the link editor builds an executable file using one shared object with a `DT_SONAME` entry of **`lib1`** and another shared object library with the path name **`/usr/lib/lib2`**, the executable file will contain **`lib1`** and **`/usr/lib/lib2`** in its dependency list.

If a shared object name has one or more slash (`/`) characters anywhere in the name, such as **`/usr/lib/lib2`** above or **`directory/file`**, the dynamic linker uses that string directly as the path name. If the name has no slashes, such as `lib1` above, three facilities specify shared object path searching, with the following precedence.

- First, the dynamic array tag `DT_RPATH` may give a string that holds a list of directories, separated by colons (`:`). For example, the string `/home/dir/usr/lib:/home/dir2/usr/lib:` tells the dynamic linker to search first the directory **`/home/dir/lib`**, then **`/home/dir2/usr/lib`**, and then the current directory to find dependencies.
- Second, a variable called `LD_LIBRARY_PATH` in the process environment may hold a list of directories as above, optionally followed by a semicolon (`;`) and another directory list. The following values would be equivalent to the previous example:

```
LD_LIBRARY_PATH=/home/dir/usr/lib:/home/dir2/usr/lib:
LD_LIBRARY_PATH=/home/dir/usr/lib;/home/dir2/usr/lib:
LD_LIBRARY_PATH=/home/dir/usr/lib:/home/dir2/usr/lib:;
```

All LD_LIBRARY_PATH directories are searched after those from DT_RPATH. Although some programs (such as the link editor) treat the lists before and after the semicolon differently, the dynamic linker does not. Nevertheless, the dynamic linker accepts the semicolon notation, with the semantics described above.

- Finally, if the other two groups of directories fail to locate the desired library, the dynamic linker searches **/usr/lib**.

For security, the dynamic linker ignores environmental search specifications (such as LD_LIBRARY_PATH) for set-user and set-group ID programs. It does, however, search DT_RPATH directories and **/usr/lib**.

Link Map

The dynamic linker creates and maintains a linked list of link map entries to describe the address space of a program using dynamic linking. The first entry in the list describes the executable file; subsequent entries describe the shared objects used by the program. The order of the link map entries is the result of performing the following conceptual algorithm. The list of link map entries is initialized to contain only the entry for the executable file. For each entry on the list (in order), the dynamic linker scans the corresponding object's _DYNAMIC section (in order) and, for each previously unreferenced shared object named by a DT_NEEDED entry, appends a link map entry for that object to the list. The result is a breadth-first linearization of the graph of shared object dependencies.

The structure of a link map entry is as follows.

```
struct link_map {
    unsigned long    l_addr;
    char             *l_name;
    Elf32_Dyn        *l_ld;
    struct link_map  *l_next;
    struct link_map *l_prev;
};
```

`l_addr` For a shared object, this is the base address of the shared object. This field is zero for an executable file.

`l_name` For a shared object, this is the virtual address of the path name of that object (e.g., **/usr/lib/libc.so.1**).

`l_name` This is the virtual address of the _DYNAMIC structure of the object.

`l_next` This is the virtual address of the next link map entry. For the last object on the chain, this field contains a null pointer.

`l_prev` This is the virtual address of the previous link map entry. For the first object on the chain, this field contains a null pointer.

Global Offset Table

Position-independent code cannot, in general, contain absolute virtual addresses. Global offset tables hold absolute addresses in private data, thus making the addresses available without compromising the position-independence and "sharability" of a program's text. A program can reference its global offset table in several ways:

- An executable file can reference its global offset table absolutely, as it would any data, because the address of the global offset table is known to the link editor.
- A shared object can reference its global offset table with position-independent references, because all of the text and data of a shared object file remains fixed relative to itself no matter where the shared object segments are assigned in memory.
- A shared object typically references its global offset table relative to the shared object's addressing base. The link editor establishes the addressing base and the location of the global offset table, so it can calculate constant offsets to global offset table entries. The addressing base value can be computed by a function in a shared object in a position-independent manner.
- References from a shared object's procedure linkage table to the global offset table procedure entries are made absolutely. This is possible because the procedure linkage table is private to the shared object.

Initially, the global offset table holds information as required by its relocation entries (see "Relocation" on page 22-27). After the dynamic linker creates memory segments for a loadable object file, it processes the relocation entries, some of which will refer to the global offset table. The dynamic linker determines the associated symbol values, calculates their absolute addresses, and sets the appropriate memory table entries to the proper values. Although the absolute addresses are unknown when the link editor builds an object file, the dynamic linker knows the addresses of all memory segments and can thus calculate the absolute addresses of the symbols contained therein.

If a program requires direct access to the absolute address of a symbol, that symbol will have a global offset table entry. Because the executable file and shared objects have separate global offset tables, a symbol's address may appear in several tables. The dynamic linker processes all the global offset table relocations before giving control to any code in the process image, thus ensuring the absolute addresses are available during execution.

The table's entry zero is reserved to hold the address of the dynamic structure, referenced with the symbol `_DYNAMIC`. This allows a program, such as the dynamic linker, to find its own dynamic structure without having yet processed its relocation entries. This is especially important for the dynamic linker, because it must initialize itself without relying on other programs to relocate its memory image.

A global offset table entry provides direct access to the absolute address of a symbol, without compromising position independence and sharability. Because the executable file and shared objects have separate global offset tables, a symbol may appear in several tables. The dynamic linker processes all the global offset table relocations giving control to any code in the process image, thus ensuring the absolute addresses are available during execution.

The dynamic linker may choose different memory segment addresses for the same shared object in different programs; it may even choose different library addresses for different executions of the same program. Nonetheless, memory segments do not change addresses once the process image is established. As long as a process exists, its memory segments reside at fixed virtual addresses.

Global offset table ("GOT") entries are created by the link editor in response to the use of certain relocation types. A GOT entry is 4 bytes long and 4-byte aligned and is allocated in writable memory private to the executable or shared object file. After relocation by the link editor, the dynamic linker, or both, a GOT entry generally contains the value of its associated symbol, which is usually the address of the entity (object or function) represented by the symbol. The one exception is the case of a function for which there is a PLT entry in the executable file. In this case the GOT entry contains the address of that PLT entry. In this way, the address by which the executable file knows the function (its PLT entry address) is also the address by which all shared objects know the function.

More efficient access to functions is provided by special GOT entries known as "global offset table procedure" ("GOTP") entries. Like GOT entries, GOTP entries are created by the link editor in response to use of certain relocation types, are 4 bytes long and 4-byte aligned, are allocated in writable memory private to the executable or shared object file, and are relocated by the link editor, dynamic linker, or both. A GOTP entry, however, may only refer to a function. During execution, the GOTP entry contains an address to which control can be transferred in order to reach the function represented by the symbol associated with the GOTP entry. Moreover, the contents of the GOTP entry may change during execution. This is "lazy binding", described below. Although the contents of a GOTP entry may change during execution, every value contained in a GOTP entry serves to transfer control correctly to the associated function.

A GOTP entry has an associated relocation of type `R_PPC_GOTP_ENT`. The relocation information and the initial contents of the entry are described under the `R_PPC_GOTP_ENT` relocation type.

The dynamic linker may perform one of two separate relocation operations for a GOTP entry. The first, called "pre-binding," is performed during the dynamic linker's relocation phase when lazy binding is in effect (when the `LD_BIND_NOW` environment variable is missing or null). In pre-binding, the dynamic linker rewrites the GOTP entry so that calling through it invokes the dynamic linker. When the first invocation is made through the GOTP entry, the dynamic linker gains control and performs the second relocation operation on the GOTP entry, called "binding." Binding involves locating the relocation table entry associated with the GOTP entry, looking up the associated symbol to find where the function resides in memory, rewriting the GOTP entry to point directly to the function, and finally transferring control to the function. If lazy binding is not in effect (the value of the `LD_BIND_NOW` environment is non-null), the dynamic linker simply performs the binding operation during its relocation phase, bypassing the pre-binding step altogether.

Lazy binding generally improves overall application performance, because unused symbols do not incur the dynamic linking overhead. Nevertheless, two situations make lazy binding undesirable for some applications. First, the initial reference to a shared object function takes longer than subsequent calls, because the dynamic linker intercepts the call to resolve the symbol. Some applications cannot tolerate this unpredictability. Second, if an error occurs and the dynamic linker cannot resolve the symbol, the dynamic linker will terminate the program. Under lazy binding, this might occur at arbitrary times. Once again, some applications cannot tolerate this unpredictability. By turning off lazy binding, the dynamic linker forces the failure to occur during process initialization.

The link editor and the dynamic linker collaborate to support lazy binding. For each GOTP entry, the link editor creates a "GOT binding" entry, a sequence of instructions that serves to transfer control to the dynamic linker. When lazy binding is in effect, the dynamic linker stores the address of the GOTP binding entry in the GOTP entry. (The addend in the relocation entry for the GOTP entry locates the GOTP binding entry.) The dynamic linker also stores a word identifying the executable or shared object file and the address of its binding routine in the second and third words, respectively, of the three words located by the `DT_PLTGOT` value for the executable or shared object file.

The GOTP binding entry is responsible for transferring control to the address contained in the word at "`DT_PLTGOT` value"+8, having extended the stack by 16 bytes with the following values:

Table 22-37. GOTP Binding Entry Stack Frame

r31 Offset	Contents
12	return address value at time of call
8	`reloc_off` value
4	word at "`DT_PLTGOT` value" + 4
0	the value 0

The `reloc_off` value is the offset, in bytes, from the `DT_JMPREL` value for the executable or shared object file containing the GOTP entry, to the relocation entry for the GOTP entry.

The GOTP binding entry may destroy the contents of certain registers. The GOTP binding entry, in transferring to the dynamic linker, must place an appropriate return address in the return address register, to maintain a proper return address chain for text description information purposes.

There are many ways for the link editor to satisfy the above requirements. One possible implementation of the GOTP binding entry is:

Table 22-38. GOTP Binding Entry

addis	r13,r0,uhi16(reloc_off)
ori	r13,r13,lo16(reloc_off)
b	GOTP_binding_helper

where `GOTP_binding_helper` is a sequence of instructions particular to the given executable or shared object file. A GOTP binding helper routine that cooperates with GOTP binding entries as shown above could be:

Table 22-39. GOTP Binding Helper

```
        addic   r1,r1,-16
        mfspr   r14,LR
        stw     r14,12(r1)
        stw     r13,8(r1)
        bl      here
here:   addis   r13,r0,uhi16(DT_PLTGOT-here)
        ori     r13,r13,lo16(DT_PLTGOT-here)
        mfspr   r14,LR
        add     r13,r13,r14
        lwz     r14,4(r13)
        stw     r14,4(r1)
        lwz     r13,8(r13)
        stw     0,0(r1)
        mtspr   CTR,r13
        bctr
```

The expression "`DT_PLTGOT-here`" represents the distance from label `here` to the `DT_PLTGOT`-specified value.

The example sequences shown for the GOTP binding entry and GOTP binding helper routine are designed not to require any relocation by the dynamic linker. Hence, they can be part of the normal text of a shared object. In particular, they don't need to reside along with PLT entries in the PLT region. However, it is convenient for the link editor to create a procedure linkage table consisting of the GOTP binding helper routine followed by PLT and GOTP binding entries for each GOTP entry.

Function Addresses

References to the address of a function from an executable file and the shared objects associated with it might not resolve to the same value. References from within shared objects will normally be resolved by the dynamic linker to the virtual address of the function itself. References from within the executable file to a function defined in a shared object will normally be resolved by the link editor to the address of the procedure linkage table entry for that function within the executable file.

To allow comparisons of function addresses to work as expected, if an executable file references a function defined in a shared object, the link editor will place the address of the procedure linkage table entry for that function in its associated symbol table entry. The dynamic linker treats such symbol table entries specially. If the dynamic linker is search-

ing for a symbol, and it encounters a symbol table entry for that symbol in the executable file, it normally follows the rules below.

- If the `st_shndx` member of the symbol table entry is not `SHN_UNDEF`, the dynamic linker has found a definition for the symbol and uses its `st_value` member as the symbol's address.
- If the `st_shndx` member is `SHN_UNDEF` and the symbol is of type `STT_FUNC` and the `st_value` member is not zero, the dynamic linker recognizes this entry as special and uses the `st_value` member as the symbol's address.
- Otherwise, the dynamic linker considers the symbol to be undefined within the executable file and continues processing.

Some relocations are associated with procedure linkage table entries. These entries are used for direct function calls rather than for references to function addresses. These relocations are not treated in the special way described above because the dynamic linker must not redirect procedure linkage table entries to point to themselves.

Procedure Linkage Table

The procedure linkage table is a repository for short sequences of code that provide convenient access to GOTP entries. A procedure linkage table ("PLT") entry is a sequence of instructions that passes control on to a procedure identified by a particular GOTP entry. The benefit of a PLT entry is that it provides an address (the address of its first instruction) to which control can simply be transferred (as by a `bsr` instruction, for example) in order to invoke a GOTP entry with the appropriate protocol.

It is usually better to access a GOTP entry directly rather than indirectly through a PLT entry. However, there are some situations in which a PLT entry can be useful.

- When code is compiled for inclusion in an executable file (and, in particular, not for inclusion in a shared object), it is generally best to compile a call into simply a `bsr` instruction, under the assumption that most calls from outside of all shared objects will be to procedures that are not in a shared object. If it turns out for such a call that the procedure being called is in a shared object, a PLT entry can be created by the link editor, and the `bsr` instruction can simply be adjusted to reference the PLT entry.
- When code is compiled for inclusion in a shared object, the compiler can emit instructions to access the GOTP entry directly. It may be useful, however, for either convenience of the compiler or compactness of the call (when many are made statically to the same GOTP entry), to use simply a `bsr` instruction and a PLT entry. The procedure linkage table is unlike a normal table in one respect--its entries are not necessarily all the same size. (Nevertheless, typically the entries will all be the same size.) The form of a typical PLT entry, for a hypothetical procedure named "name", is shown below, as if it were written in assembly language.

Table 22-40. PLT Entry

name:	addis	r13,r0,hi16(name@gotp)
	lwz	r13,lo16(name@gotp)(r13)
	mtspr	CTR,r13
	bctr	

Although the instruction sequence shown above is only one of many possible sequences, the following points will invariably be true:

- The GOTP entry for the procedure is referenced absolutely. Because the global offset table for a shared object may reside at different locations in different processes, the PLT entry code cannot be shared by different processes.
- Register `r13` is used to load the contents of the GOTP entry.
- No general purpose register other than `r13` or `r11` is changed by the PLT entry sequence.

Executable files and shared object files have separate procedure linkage tables, just as they have separate global offset tables. The treatment by the link editor and dynamic linker can vary in two different cases. The procedure linkage table in an executable file can be relocated by the link editor, so it can be placed in the text area and shared by all processes executing that file. Note that, in this case, the dynamic linker doesn't act on the procedure linkage table at all. Because the PLT entry refers to absolute addresses in the global offset table, however, the procedure linkage table in a shared object file cannot be relocated until the shared object has had its memory assigned by the dynamic linker. In the shared object case, the link editor constructs the procedure linkage table in a segment that is initially writable but not executable. The link editor records the extent of the PLT region with the `DT_PPC_PLTSTART` and `DT_PPC_PLTEND` information. The dynamic linker loads the shared object, performs relocations (including those on the procedure linkage table), then uses `mprotect(KE_OS)` to change the segment containing the procedure linkage table from writable to executable. Note that the area of memory subject to being changed from writable to executable is the area containing the PLT region, rounded outward on each end to a 64K boundary.

Hash Table

A hash table of `Elf32_Word` objects supports symbol table access. Labels in Table 22-41 help explain the hash table organization, but they are not part of the specification.

Table 22-41. Symbol

nbucket
nchain
bucket[0] ... bucket[nbucket - 1]
chain[0] ... chain[nchain - 1]

The `bucket` array contains `nbucket` entries, and the `chain` array contains `nchain` entries; indexes start at 0. Both `bucket` and `chain` hold symbol table indexes. Chain table entries parallel the symbol table. The number of symbol table entries should equal `nchain`; so symbol table indexes also select chain table entries. A hashing function (shown below) accepts a symbol name and returns a value that may be used to compute a `bucket` index. Consequently, if the hashing function returns the value `x` for some name, `bucket[x%nbucket]` gives an index, `y` into both the symbol table and the chain table. If the symbol table entry is not the one desired, `chain[y]` gives the next symbol table entry with the same hash value. One can follow the `chain` links until either the selected symbol table entry holds the desired name or the `chain` entry contains the value `STN_UNDEF`.

```
unsigned long
elf_hash(const unsigned char *name)
{
    unsigned long h = 0, g;

    while (*name)
    {
        h = (h << 4) + *name++;
        if (g = h & 0xf0000000)
            h ^= g >> 24;
        h &= ~g;
    }
    return h;
}
```

Initialization and Termination Functions

After the dynamic linker has built the process image and performed the relocations, each shared object gets the opportunity to execute some initialization code. These initialization functions are called in no specified order, but all shared object initializations happen before the executable file gains control. Similarly, shared objects may have termination functions, which are executed with the `atexit()` mechanism after the base process begins its termination sequence. (See **`atexit(3C)`**.) Once again, the order in which the dynamic linker calls termination functions is unspecified. Shared objects designate their initialization and termination functions through the `DT_INIT` and `DT_FINI` entries in the

dynamic structure, described in “Dynamic Section” on page 22-47. Typically, the code for these functions resides in the **.init** and **.fini** sections, mentioned in “Section Header” on page 22-9.

Although the `atexit()` termination processing normally will be done, it is not guaranteed to have executed upon process death. In particular, the process will not execute the termination processing if it calls `_exit()` or if the process dies because it received a signal that it neither caught nor ignored.

Symbolic Debugging Information

ELF does not specify a format for representation of symbolic debugging information. Systems vendors are free to provide a representation of their choice. The Concurrent C, Fortran, and Ada compilers produce DWARF symbolic debugging information as described in the DWARF version 2 draft 6 specification (See Chapter 24 (“DWARF Debugging Information Format”).) with the exceptions noted below.

Several attributes have been added to support Concurrent Fortran 77 extensions. These attributes are described below.

`AT_datapool` The presence of an `AT_datapool` flag in a `TAG_common_block` DIE indicates that the `TAG_common_block` DIE is actually a data pool. The DIEs owned by that data pool are `TAG_variable` DIEs rather than `TAG_member` DIEs.

`AT_pointer_block` (See `AT_pointer_base` below)

`AT_pointer_base` The presence of an `AT_pointer_block` flag in a `TAG_common_block` DIE indicates that the `TAG_common_block` DIE is actually a pointer block. Each pointer block DIE also contains an `AT_pointer_base` attribute which is a reference to the `TAG_variable` or `TAG_member` which holds the block’s base address. Each `TAG_member` DIE owned by the pointer block contains an `AT_data_member_location` which evaluates to the offset of the member from the base address.

To facilitate the access of DWARF symbolic debugging information, some systems vendors provide a library of routines which may be used by a user’s program. PowerUX provides **/usr/ccs/lib/libdwarf.a**, which complies to DWARF Access Library specification, version 1, draft 1. (See Chapter 25 (“DWARF Access Library (libdwarf)”).), with the following exceptions.

The type information query operations have been modified to more closely map to the DWARF version 2 draft 6 specification. The functions described in section 5.5 of this specification have been replaced with the following:

5.5 Type Information Query Operations

These operations return information concerning data types.

```
Dwarf_Signed dwarf_modtags (
    Dwarf_Type  typ,
    Dwarf_Half  **tagbuf,
    Dwarf_Error *error)
```

The function `dwarf_modtags()` sets `tagbuf` to point to an array of modifier tags represented by the `Dwarf_Type` descriptor `typ` and returns the number of elements in the array; `DLV_NOCOUNT` is returned on error. The storage pointed to by `tagbuf` after a successful return of `dwarf_modtags()` should be free'd when no longer of interest (see `dwarf_dealloc()`).

```
Dwarf_Bool dwarf_isbasetype (
    Dwarf_Type  typ,
    Dwarf_Error *error)
```

The function `dwarf_isbasetype()` returns non-zero if the `Dwarf_Type` descriptor `typ` represents a base type; zero is returned otherwise.

```
Dwarf_Die dwarf_basetype (
    Dwarf_Type  typ,
    Dwarf_Error *error)
```

The function `dwarf_basetype()` returns a `Dwarf_Die` descriptor that represents the base type of the type represented by the descriptor `typ`; `NULL` is returned if type does not represent a base type or an error occurred.

```
char* dwarf_base_name (
    Dwarf_Die   base,
    Dwarf_Error *error)
```

The function `dwarf_base_name()` returns a pointer to a `NULL` terminated string of characters that represents the name of the base type represented by the `Dwarf_Die` descriptor base; `NULL` is returned on error.

```
Dwarf_Small dwarf_base_encoding (
    Dwarf_Die   base,
    Dwarf_Error *error)
```

The function `dwarf_base_encoding()` returns a code representing the encoding of the base type represented by the `Dwarf_Die` descriptor base; 0 is returned on error.

```
Dwarf_Signed dwarf_base_size (
    Dwarf_Die   base,
    Dwarf_Error *error)
```

The function `dwarf_base_size()` returns an integer representing the size of the base type represented by the `Dwarf_Die` descriptor base; `DLV_NOCOUNT` is returned on error.

```
Dwarf_Die dwarf_udtype (
    Dwarf_Type  typ,
    Dwarf_Error *error)
```

The function `dwarf_udtype()` returns the `Dwarf_Die` descriptor representing the user-defined type given by the `Dwarf_Type` descriptor `typ`; `NULL` is returned on error or if `typ` is not a user-defined type.

23
tdesc Information

23
tdesc Information

Introduction

In order to obtain meaningful stack walkbacks (traces) when debugging programs, these programs need information describing the text of the various modules in the program. The 88open Object Compatibility Standard (OCS) defines this information, calling it *text description* (or *tdesc*) information. This information also permits an Ada program to perform correct exception handling. Concurrent compilation systems for the supporting hardware platforms produce and use this information.

Every module of code has one or more accompanying blocks of information called *tdesc chunks*. These chunks will be described below. Compilers--C, Fortran, and Ada--automatically produce these chunks. Assembly writers must provide these chunks in their assembly code. The link editor combines the tdesc chunks from the various modules to be link edited, and the link edited executable program contains the final collection of tdesc chunks. The portion of text described by a *tdesc chunk* is known as a *text chunk*.

A body of code consists of three general parts:

prologue — Establishes the stack frame pointer, saves any registers that need to be saved in the stack frame, and saves the return address in the stack frame, when necessary.

epilogue — Adjusts the stack pointer to its incoming value and returns to the caller.

procedure body — Is the unchanging part of the code (with respect to the above actions).

Figure 23-1 illustrates these parts.

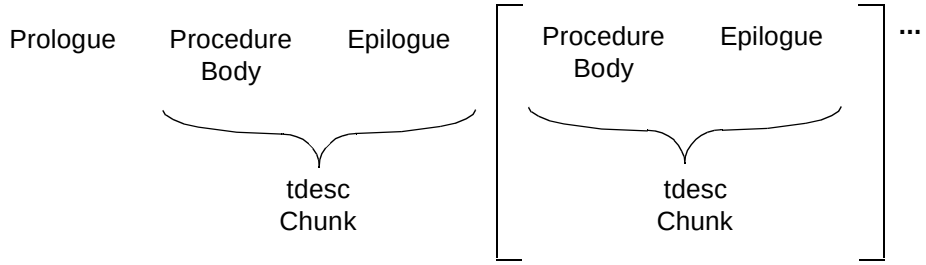


Figure 23-1. The Parts of a Body of Code

Currently, tdesc information is meaningful only for the procedure body. Epilogue code can be interspersed through the procedure body, however, so the tdesc information described below is usually adequate for covering epilogue code as well. Often, a single tdesc chunk describes an entire function or module. If epilogue code (or any code which modifies the stack frame pointer) is placed in the middle of the procedure body, then multiple tdesc chunks are needed to ensure none of their coverages include the epilogue code. No more than one tdesc chunk may describe the same piece of code. (I.e., text chunks may not overlap.)

tdesc information is specified in assembly language as part of the **`.tdesc`** section. The assembly syntax for assembling into this section is:

```
        section   .tdesc,"x"
```

A series of words is assembled into this section to define the chunk.

tdesc Chunks

The header file **`tdesc.h`** provides declarations and definitions for the various components of tdesc information. The C structure definition for a tdesc chunk is as follows.

```
struct __tdesc_chunk {
    unsigned int zeroes:8;    /* zeroes */
    unsigned int length:22;   /* info length */
    unsigned int alignment:2; /* alignment exponent */
    int protocol_number;      /* protocol number */
    int *start;   /* start address of text chunk */
    int *end;     /* end address of text chunk */
};
```

tdesc Chunk Declaration:

Field Name	Contents
`zeroes`	This member contains a zero.
`length`	This member gives the length in bytes of the tdesc chunk.
`alignment`	This member gives the required alignment of the info protocol contained in the chunk.

Value	Byte Alignment
0	1
1	2
2	4
3	8

`protocol_number` This member identifies the protocol associated with the chunk.

Name	Value
`_INFO_GENERAL_PROTOCOL_NUMBER`	1
`_INFO_PIC_PROTOCOL_NUMBER`	2
`_INFO_EXTENDED_PROTOCOL_NUMBER`	3
`_INFO_PIC_EXTENDED_PROTOCOL_NUMBER`	4
`_INFO_EXCEPTION_PROTOCOL_NUMBER`	0x7f
`_INFO_PIC_EXCEPTION_PROTOCOL_NUMBER`	0x7e
`_INFO_EXTENDED_EXCEPTION_PROTOCOL_NUMBER`	0x7d
`_INFO_PIC_EXTENDED_EXCEPTION_PROTOCOL_NUMBER`	0x7c

Substrings in the preceding names identify the kind of protocol.

`GENERAL` This is the standard protocol. The following protocols are extensions to this protocol.

`PIC` This protocol provides position-independent starting and ending addresses, for use in shared objects.

`EXTENDED` This protocol provides information on features unique to particular systems. Presently, this includes the floating-point registers which are saved by the text chunk.

`EXCEPTION` This protocol provides a pointer to an Ada exception table.

`start` This member contains a pointer to the starting address (inclusive) of the corresponding text chunk. The address is after the prologue code and at the start of the procedure body.

`end` This member contains a pointer to the ending address (exclusive) of the corresponding text chunk. The address is immediately after the procedure body.

Following these initial four words of the tdesc chunk is the data for the chunk's *info protocol*. Some protocols may have more than one variant. The C union/structure definition for the various info protocols is as follows.

```
union ___info_protocol {
     struct __general_protocol {
          unsigned int variant:8;        /* info variant */
          unsigned int unused_1:5;       /* reserved for future use */
          unsigned int save_mask:13;     /* register save mask */
          unsigned int discriminant:1;   /* return address info
                                             discriminant */
          unsigned int frame_register:5; /* frame address register */
          int frame_offset;              /* frame address offset */
          int return_info;               /* return address info */
          int save_offset;               /* register save offset */
     } general;

     struct __extended_protocol {
          unsigned int variant:8;        /* info variant */
          unsigned int unused_1:5;       /* reserved for future use */
          unsigned int save_mask:13;     /* register save mask */
          unsigned int discriminant:1;   /* return address info
                                            discriminant*/
          unsigned int frame_register:5; /* frame address register */
          int frame_offset;              /* frame address offset */
          int return_info;               /* return address info */
          int save_offset;               /* register save offset */
          unsigned int extended_save_mask:10;/* floating register
                                                save mask */
          unsigned int unused_2:22;      /* reserved for future use*/
     } extended;

     struct __exception_protocol {
          unsigned int variant:8;        /* info variant */
          unsigned int unused_1:5;       /* reserved for future use */
          unsigned int save_mask:13;     /* register save mask */
          unsigned int discriminant:1;   /* return address info
                                             discriminant */
          unsigned int frame_register:5; /* frame address register */
          int frame_offset;              /* frame address offset */
          int return_info;               /* return address info */
          int save_offset;               /* register save offset */
          int *ada_entry;                /* start of prologue pointer*/
          int *ada_exception;            /* Ada exception pointer */
     } exception;

     struct __extended_exception_protocol {
          unsigned int variant:8;        /* info variant */
          unsigned int unused_1:5;       /* reserved for future use */
          unsigned int save_mask:13;     /* register save mask */
          unsigned int discriminant:1;   /* return address info
                                             discriminant */
          unsigned int frame_register:5; /* frame address register */
          int frame_offset;              /* frame address offset */
          int return_info;               /* return address info */
          int save_offset;               /* register save offset */
          int *ada_entry;                /* start of prologue pointer*/
```

```
            int *ada_exception;           /* Ada exception pointer */
            unsigned int extended_save_mask:10;/* floating register
                                                  save mask */
            unsigned int unused_2:22;     /* reserved for future use */
        } extended_exception;
        struct __full_save_protocol {
            unsigned int variant:8;       /* info variant */
            unsigned int frame_register:5; /* frame address register */
            unsigned int indirect:1;      /* interpretation of
                                             save_offset */
            unsigned int mask ;           /* mask defining saved
                                             registers */
            int frame_offset;             /* frame address offset */
            int save_offset;              /* register save offset */
            int *ada_exception;           /* pointer to exception handler */
          } ;
        struct __info_indirect_protocol {
            unsigned int variant:8;       /* info variant */
            int *alternate_pc ;           /* alternate PC address */
          } ;
    }
```

Info Protocols Declaration:

Field Name **Contents**

`variant` This member identifies the variant of the info protocol.

Name	Value
_INFO_GENERAL_VARIANT	1
_INFO_EXTENDED_VARIANT	3
_INFO_EXCEPTION_VARIANT	0x7f
_INFO_EXTENDED_EXCEPTION_VARIANT	0x7d
_INFO_SIGACTHANDLER_VARIANT	0x7b
_INFO_FULL_SAVE_VARIANT	0x71
_INFO_INDIRECT_VARIANT	0x70

`_INFO_GENERAL_VARIANT`
This is the standard variant. Most of the following variants are extensions to this variant.

`_INFO_EXTENDED_VARIANT`
This variant provides information on features unique to particular systems. Presently, this includes the extended registers which are saved by the text chunk.

`_INFO_EXCEPTION_VARIANT`
: This variant provides a pointer to an Ada exception table.

`_INFO_EXTENDED_EXCEPTION_VARIANT`
: This variant provides information on features unique to particular systems. Presently, this includes the extended registers which are saved by the text chunk. This variant also provides a pointer to an Ada exception table.

`_INFO_SIGACTHANDLER_VARIANT`
: This variant indicates that the corresponding text chunk is in the C library `sigacthandler()` function. The prototype for this function is:

```
void _sigacthandler(int sig, siginfo_t *sip,
        ucontext_t *ucp, void (*handler)())
```

Walkback information for identifying the approximate location where the signal was raised can be determined as follows:

- The value of `ucp` is at:

  stack pointer + info_protocol.general.frame_offset + 8

- The approximate address of the instruction where the signal was raised is at:

  ucp->uc_mcontext.gregs[R_SRR0]

- The address of the stack pointer for the text chunk of the routine where the signal was raised is at:

  ucp->uc_mcontext.gregs[R_R1]

`save_mask`
: This mask generally identifies the general-purpose registers which are preserved by the corresponding text chunk in the current *stack frame*. A bit is on in `save_mask` if the corresponding register is preserved.

Bit in Word	Register
18	r2
17	r16
16	r17
...	...
6	r27

`zero`
: This field contains a zero bit.

`discriminant` This member provides information on how to determine the return address from the corresponding text chunk. If `discriminant` is 0, then the return register is the general-purpose register whose number is contained in the `return_info` member. If `discriminant` is 1, then the return address is the value of the word at the stack frame position specified by the `return_info` member.

`frame_register` This member gives the number of the general-purpose register which is used to locate the current stack frame. This register is often the stack pointer itself, but it need not be. In the prologue:

```
addi    r1,r1,-40
```

`frame_register` is r1.

`frame_offset` This member provides the value which is added to the frame register to locate the current stack frame. Often, the prologue code decrements the incoming stack pointer, providing room on the stack for local variables and arguments to functions called by the current procedure. The frame offset is usually the value that is subtracted from the incoming stack pointer. In the prologue:

```
addi    r1,r1,-40
```

`frame_offset` is 40.

`return_info` This member identifies the return-address register or the location where the return address resides within the current stack frame, as described above. Upon entry to the procedure, the return address is in register `lr`. If no other procedure calls are made from the current procedure, and if that register is not modified in the procedure body, then the return address can be found in that register. In this case, `discriminant` is 0, and `return_info` is 65 (for `lr`). If the return-address register is modified in the procedure body, then the prologue code will save it on the stack. In this case `discriminant` is 1, and `return_info` provides an offset from **(**`frame_register` + `frame_offset`**)** at which the return address can be obtained. For the prologue:

```
addi    r1,r1,-40
mflr    r13
stw     r13,48(r1)
```

`discriminant` is 1 and `return_info` is +8 or +48, relative to the caller's frame.

`save_offset` For most variants, this member gives the base offset within the current stack frame of the start of the general-purpose saved registers. It provides an offset from **(**`frame_register` + `frame_offset`**)** at which the first general-purpose register is preserved. Generally, only those registers specified in `save_mask` are saved in this area. For r16-r27, inclusive, successively higher-numbered registers are stored at successively

higher addresses within the register save area, and r2 (if it is preserved) is saved at the next higher address after the r16-r27 group.

For the `_INFO_FULL_SAVE_VARIANT`, this member gives the base offset within the current stack frame of the start of the register save area. See the discussion of `indirect` (below) for the interpretation of `save_offset`. See the discussion of `mask` (below) for information about the register save area.

`ada_entry` This member provides the address of the start of the prologue.

`ada_exception` This member provides the address of an Ada exception table pointer.

`extended_save_mask`
This mask generally identifies the floating-point registers which are preserved by the corresponding text chunk in the current stack frame. A bit is on in the mask if the corresponding register is preserved.

Bit in Word	Register
31	f22
30	f23
...	...
22	f31

. Generally, the preserved floating-point registers immediately precede the preserved general-purpose registers, with alignment to the next 16-byte boundary. For all variants, successively higher-numbered registers are stored at successively higher addresses within the register save area.

`unused_1` This member is reserved for future use.

`unused_2` This member is reserved for future use.

`indirect` This member indicates how `save_offset` is to be interpreted. If `indirect` is 0, `save_offset` is the offset from `(frame_register + frame_offset)` of the register save area. If `indirect` is 1, `save_offset` is the offset from `(frame_register + frame_offset)` of the word containing a byte pointer to the register save area.

`mask` This mask identifies which registers are saved in the stack frame. A bit is on in `mask` if the corresponding register(s) is/are saved. The bits are

Name	Value	Registers
`_INFO_FULL_SAVES_FPSCR`	0x1	floating-point status and control register (fpscr)
`_INFO_FULL_SAVES_FPREGS`	0x2	floating-point registers (f0-f31)
`_INFO_FULL_SAVES_CR`	0x4	condition register (cr)
`_INFO_FULL_SAVES_XER`	0x8	integer exception register (xer)
`_INFO_FULL_SAVES_LR`	0x10	link register (lr)
`_INFO_FULL_SAVES_CTR`	0x20	count register (ctr)

The register save area for the full-save protocol is laid out as follows. Note that even if optional registers do not contain meaningful information, they still have space allocated for them.

r0-r31
f0-f31 (optional)
cr (optional)
reserved word
pc
xer (optional)
ctr (optional)
lr (optional)
reserved word
fpscr (optional)

`alternate_pc` This member identifies an alternate PC value that should be used for locating the actual tdesc information for this code.

tdesc in Executable Programs and Shared Objects

PowerUX provides facilities for the creation and execution of both statically linked programs and dynamically linked programs. A statically linked program contains all of the code and data in the on-disk image of the program. A dynamically linked program consists of a statically linked portion, which is the on-disk image of the program, and one or more shared objects which are dynamically linked into the process' address space during execution of the program. (See Chapter 4 ("Link Editor and Linking") and Chapter 22 ("Executable and Linking Format (ELF)") for additional information on shared objects and dynamic linking.)

The link editor concatenates tdesc chunks from the object files which constitute an object, whether it be the statically linked portion of a program or a shared object. These concatenated tdesc chunks reside in a separate **`.tdesc`** section of the object.

Two linker-provided protocols describe and locate the body of tdesc chunks. The first is the *debug info protocol*. A C structure definition for it is as follows.

```
struct __debug_info_protocol {
    int protocol_number; /* protocol number */
    int tdesc;           /* pointer to map protocol */
    int number_text;     /* number of text words */
    int *text_words;     /* pointer to text words */
    int number_data;     /* number of data words */
    int *data_words;     /* pointer to data words */
};
```

Debug Info Protocol Declaration:

Field Name	Contents
`protocol_number`	This member identifies the particular debug info protocol. The `_DEBUG_INFO_PROTOCOL_NUMBER` protocol has the value 1.
`tdesc`	This member provides the virtual address of the map protocol.
`number_text`	This member indicates how many words are available in the text segment for use by debuggers.
`text_words`	This member provides the virtual address of the available words in the text segment.
`number_data`	This member indicates how many words are available in the data segment for use by debuggers.
`data_words`	This member provides the virtual address of the available words in the data segment.

If the symbol table is present in the program, the value of the symbol `_debug_info` is the virtual address of the debug info protocol. Both the program header and the section header provide this address.

The *map protocol* locates and gives the lengths of the concatenated tdesc chunks. Concurrent compilation systems provide two different map protocols. The C union/structure definition for them is as follows.

```
union __map_protocol {
    struct __minimal_protocol {
        int protocol_number;      /* protocol number */
        int tdesc_end;            /* address beyond end
                                     of tdesc chunks */
    } minimal ;

    struct __pointer_protocol {
        int protocol_number;      /* protocol number */
        int tdesc_end;            /* address beyond end
                                     of tdesc chunks */
        int pointer_array_length; /* length of pointer
                                     array */
        int filler;               /* filler for 8-byte
                                      boundary alignment */
    } pointer ;
};
```

Map Protocols Declaration:

Field Name **Contents**

`protocol_number` This member identifies the particular map protocol.

Name	Value
`_MAP_MINIMAL_PROTOCOL_NUMBER`	1
`_MAP_POINTER_PROTOCOL_NUMBER`	0x10001

`_MAP_MINIMAL_PROTOCOL_NUMBER`
: This is the standard protocol. In this protocol, the tdesc chunks are concatenated together in an arbitrary order immediately after the map protocol.

`_MAP_POINTER_PROTOCOL_NUMBER`
: In this protocol, the tdesc chunks are concatenated together in an arbitrary order, but a sorted array of pointers allows debuggers to locate a particular tdesc chunk through a binary search of starting addresses. The sorted array of pointers immediately follows the first four words of this protocol, and the concatenated tdesc chunks immediately follow the array.

  An array element provides both the virtual address of a tdesc chunk and the virtual starting address of the corresponding text chunk. The array is sorted in increasing order of the virtual starting addresses of the corresponding text chunks. The C structure definition of an array element is as follows.

```
struct __tdesc_pointer {
int *start;     /* start address of
                   text chunk */
int *tdesc;     /* address of
                   tdesc chunk */
};
```

Pointer Array Declaration:

`start` This member provides the virtual starting address of the text chunk.

`tdesc` This member provides the virtual starting address of the corresponding tdesc chunk.

`tdesc_end` This member gives the virtual address of the byte immediately after the last byte of the concatenated tdesc chunks.

`pointer_array_length`
This member provides the byte length of the sorted array of pointers.

`filler` This member merely forces alignment of succeeding information.

Dynamically Linked Programs:

For a dynamically linked program, a linked list of tdesc maps identifies the tdesc information for each object (static or shared) which makes up the program. A tdesc map has the following format:

word 0: 2 (the version number)

word 1: The address of the byte immediately beyond the end of this tdesc map

words 2 through end-of-map:
An array of pointers, where an array element corresponds to an object (static or shared) which makes up the program. Each array element is a pair of words:

word i: The virtual address of the map protocol for the object

word i+1: The virtual address of the base of the object (or 0, for the static object in the program)

In the last element of a tdesc map, word i may or may not be zero. If it is zero, then there are no more tdesc maps. If it is nonzero, then it is the address of the next tdesc map in the linked list.

The first tdesc map in the list corresponds to the **`.tdesc_map2`** section in the static portion of the program. The contents of this tdesc map are supplied by the link editor and the system program interpreter (dynamic linker). The contents of any other tdesc maps are supplied by the dynamic linker, typically through invocation of **`dlopen(3X)`**,

When dynamic linking takes place for ELF programs, the link maps identify the various objects, and their base addresses, which comprise the running program. See Chapter 22 (“Executable and Linking Format (ELF)”) for more information on dynamic linking and link maps.

Special Symbols:

If a symbol table is present in an object (static or shared), the following symbols will be present:

`_tdesc`: In a statically linked program and in a shared object, the value of this symbol is the virtual address of the map protocol. In the static portion of a dynamically linked program, the value of this symbol is the virtual address of the beginning of the contents of the **`.tdesc_map2`** section.

`_debug_info`: In a statically linked program and in the static portion of a dynamically linked program, the value of this symbol is the virtual address of the debug info protocol. In a shared object, the value of this symbol is 0.

Examples

The examples that follow show C functions and corresponding assembly code for the **`.text`** and the **`.tdesc`** sections.

Examples:

```
func (a, b, c, d, e)
double a, b, c, d, e;
{
    printf (" %e ", d+e);
    proc ();
    printf (" %e ", d+e);
}
```

	addi	r1,r1,-96	`frame_offset` = 96
	stfd	f22,64(r1)	`fp save_mask` =
	stfd	f23,72(r1)	1100000000
	mflr	r13	`save_offset` = 64-96 =
	stw	r13,104(r1)	-32
	fmr	f23,f4	return_info = 104-96 = 8
	fmr	f22,f5	return discriminant = 1
@LSTART:			start of procedure body (start of text chunk)
	lis	r3,uhi16(@L10)	
	ori	r3,r3,lo16(@L10)	
	fadd	f1,f23,f22	
	bl	printf	lr is modified
	bl	proc	
	lis	r3,uhi16(@L11)	
	ori	r3,r3,lo16(@L11)	
	fadd	f1,f23,f22	
	bl	printf	
	lfd	f22,64(r1)	
	lfd	f23,72(r1)	
	lwz	r13,104(r1)	
	mtlr	r13	
	addi	r1,r1,96	
			start of epilogue
			end of procedure body end of epilogue
@LEND:			(end of text chunk)
	section	.tdesc,"x"	start of tdesc chunk
	word	0x52	
	word	0x3	
	word	@LSTART	
	word	@LEND	
	word	0x30000021	
	word	96	
	word	8	
	word	-32	
	word	0x300	end of tdesc chunk
	text		change back to text section

```
sub (a, b, c)
{
    int i;
    i = a + b + c;
}
```

---

			no prologue code no frame pointer needed no registers saved return `discriminant` = 0 `return_info` = r1
@LSTART:			text chunk = entire module
	add	r3,r3,r4	
	add	r3,r3,r5	
	blr		
@LEND:			
	section	.tdesc,"x"	
	word	0x42	
	word	0x1	
	word	@LSTART	
	word	@LEND	
	word	0x1000001	
	word	0	
	word	1	
	word	0	

```
func ()
{
    int a[70000];
    int b[70000];
    a[3] = 4;
    b[3] = 4;
    proc (&a, &b);
}
```

```
                addi    r1,r1,-16       frame_offset = 16
                stw     r2,0(r1)        save_offset = -16
                mflr    r13             return_info = 8
                stw     r13,24(r1)      stack frame pointer = r2
                mr      r2,r1
                addis   r13,r0,9
                addi    r13,r13,35776
                subfc   r1,r13,r1
                lis     r3,uhi16(0xfff77480)
                ori     r3,r3,lo16(0xfff77480)
                add     r3,r2,r3
@LSTART:                                start of procedure body
                                        (start of text chunk)
                li      r4,lo16(4)
                stw     r4,12(r3)
                li      r5,lo16(4)
                lis     r4,uhi16(0x445cc)
                ori     r4,r4,lo16(0x445cc)
                stwx    r5,r3,r4
                mr      r5,r3
                lis     r4,uhi16(0x445cc)
                ori     r4,r4,lo16(0x445cc)
                add     r4,r3,r4
                mr      r3,r5
                bl      proc
                mr      r1,r2
                lwz     r2,0(r1)
                lwz     r13,24(r1)
                mtlr    r13
                addi    r1,r1,16
                blr                     begin epilogue


                                        end of procedure body
                                        end of epilogue
@LEND:                                  (end of text chunk)
                section .tdesc,"x"
                word    0x42
                word    0x1
                word    @LSTART
                word    @LEND
                word    0x1040022
                word    16
                word    8
                word    -16
                text
```

24
DWARF Debugging Information Format

24
DWARF Debugging Information Format

Introduction

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

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

Purpose and Scope

The debugging information format described in this document is designed to meet the symbolic, source-level debugging needs of different languages in a unified fashion by requiring language independent debugging information whenever possible. Individual needs, such as C++ virtual functions or Fortran common blocks are accommodated by creating attributes that are used only for those languages. The UNIX International Programming Languages SIG believes that this document sufficiently covers the debugging information needs of C, C++, FORTRAN77, Fortran90, Modula2 and Pascal.

This document describes DWARF Version 2, the second generation of debugging information based on the DWARF format. While DWARF Version 2 provides new debugging information not available in Version 1, the primary focus of the changes for Version 2 is the representation of the information, rather than the information content itself. The basic structure of the Version 2 format remains as in Version 1: the debugging information is represented as a series of debugging information entries, each containing one or more attributes (name/value pairs). The Version 2 representation, however, is much more compact than the Version 1 representation. In some cases, this greater density has been achieved at the expense of additional complexity or greater difficulty in producing and processing the DWARF information. We believe that the reduction in I/O and in memory paging should more than make up for any increase in processing time.

Because the representation of information has changed from Version 1 to Version 2, Version 2 DWARF information is not binary compatible with Version 1 information. To make it easier for consumers to support both Version 1 and Version 2 DWARF information, the Version 2 information has been moved to a different object file section,.debug_info.

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

Overview

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

“General Description” on page 24-4 describes the overall structure of the information and attributes that are common to many or all of the different debugging information entries. “Program Scope Entries” on page 24-19, “Data Object and Object List Entries” on page 24-31, and “Type Entries” on page 24-34 describe the specific debugging information entries and how they communicate the necessary information about the source program to a debugger. “Other Debugging Information” on page 24-47 describes debugging information contained outside of the debugging information entries, themselves. The encoding of the DWARF information is presented in “Data Representation” on page 24-64.

“Future Directions” on page 24-85 describes some future directions for the DWARF specification.

In the following sections, text in normal font describes required aspects of the DWARF format. Text in italics is explanatory or supplementary material, and not part of the format definition itself.

Vendor Extensibility

This document does not attempt to cover all interesting languages or even to cover all of the interesting debugging information needs for its primary target languages (C, C++, FORTRAN77, Fortran90, Modula2, Pascal). Therefore the document provides vendors a way to define their own debugging information tags, attributes, base type encodings, location operations, language names, calling conventions and call frame instructions by reserving a portion of the name space and valid values for these constructs for vendor specific additions. Future versions of this document will not use names or values reserved for vendor specific additions. All names and values not reserved for vendor additions, however, are reserved for future versions of this document. See “Data Representation” on page 24-64 for details.

Changes from Version 1

The following is a list of the major changes made to the DWARF Debugging Information Format since Version 1 of the format was published (January 20, 1992). The list is not meant to be exhaustive.

- Debugging information entries have been moved from the .debug to the .debug_info section of an object file.
- The tag, attribute names and attribute forms encodings have been moved out of the debugging information itself to a separate abbreviations table.
- Explicit sibling pointers have been made optional. Each entry now specifies (through the abbreviations table) whether or not it has children.
- New more compact attribute forms have been added, including a variable length constant data form. Attribute values may now have any form within a given class of forms.
- Location descriptions have been replaced by a new, more compact and more expressive format. There is now a way of expressing multiple locations for an object whose location changes during its lifetime.
- There is a new format for line number information that provides information for code contributed to a compilation unit from an included file. Line number information is now in the .debug_line section of an object file.
- The representation of the type of a declaration has been reworked.
- A new section provides an encoding for pre-processor macro information.
- Debugging information entries now provide for the representation of non-defining declarations of objects, functions or types.
- More complete support for Modula2 and Pascal has been added.
- There is now a way of describing locations for segmented address spaces.
- A new section provides an encoding for information about call frame activations.

- The representation of enumeration and array types has been reworked so that DWARF presents only a single way of representing lists of items.
- Support has been added for C++ templates and exceptions.

General Description

The Debugging Information Entry

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

The set of required tag names is listed in Table 24-1. The debugging information entries they identify are described in “Program Scope Entries” on page 24-19, “Data Object and Object List Entries” on page 24-31, and “Type Entries” on page 24-34.

The debugging information entries in DWARF Version 2 are intended to exist in the .debug_info section of an object file.

Table 24-1. Tag Names

DW_TAG_access_declaration	DW_TAG_array_type
DW_TAG_base_type	DW_TAG_catch_block
DW_TAG_class_type	DW_TAG_common_block
DW_TAG_common_inclusion	DW_TAG_compile_unit
DW_TAG_const_type	DW_TAG_constant
DW_TAG_entry_point	DW_TAG_enumeration_type
DW_TAG_enumerator	DW_TAG_file_type
DW_TAG_formal_parameter	DW_TAG_friend
DW_TAG_imported_declaration	DW_TAG_inheritance
DW_TAG_inlined_subroutine	DW_TAG_label
DW_TAG_lexical_block	DW_TAG_member
DW_TAG_module	DW_TAG_namelist
DW_TAG_namelist_item	DW_TAG_packed_type
DW_TAG_pointer_type	DW_TAG_ptr_to_member_type
DW_TAG_reference_type	DW_TAG_set_type
DW_TAG_string_type	DW_TAG_structure_type
DW_TAG_subprogram	DW_TAG_subrange_type

Table 24-1. Tag Names (Cont.)

DW_TAG_subroutine_type	DW_TAG_template_type_param
DW_TAG_template_value_param	DW_TAG_thrown_type
DW_TAG_try_block	DW_TAG_typedef
DW_TAG_union_type	DW_TAG_unspecified_parameters
DW_TAG_variable	DW_TAG_variant
DW_TAG_variant_part	DW_TAG_volatile_type
DW_TAG_with_stmt	

Attribute Types

Each attribute value is characterized by an attribute name. The set of attribute names is listed in Table 24-2.

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

Table 24-2. Attribute Names

DW_AT_abstract_origin	DW_AT_accessibility
DW_AT_address_class	DW_AT_artificial
DW_AT_base_types	DW_AT_bit_offset
DW_AT_bit_size	DW_AT_byte_size
DW_AT_calling_convention	DW_AT_common_reference
DW_AT_comp_dir	DW_AT_const_value
DW_AT_containing_type	DW_AT_count
DW_AT_data_member_location	DW_AT_decl_column
DW_AT_decl_file	DW_AT_decl_line
DW_AT_declaration	DW_AT_default_value
DW_AT_discr	DW_AT_discr_list
DW_AT_discr_value	DW_AT_encoding
DW_AT_external	DW_AT_frame_base
DW_AT_friend	DW_AT_high_pc

Table 24-2. Attribute Names (Cont.)

DW_AT_identifier_case	DW_AT_import
DW_AT_inline	DW_AT_is_optional
DW_AT_language	DW_AT_location
DW_AT_low_pc	DW_AT_lower_bound
DW_AT_macro_info	DW_AT_name
DW_AT_namelist_item	DW_AT_ordering
DW_AT_priority	DW_AT_producer
DW_AT_prototyped	DW_AT_return_add
DW_AT_segment	DW_AT_sibling
DW_AT_specification	DW_AT_start_scope
DW_AT_static_link	DW_AT_stmt_list
DW_AT_stride_size	DW_AT_string_length
DW_AT_type	DW_AT_upper_bound
DW_AT_use_location	DW_AT_variable_parameter
DW_AT_virtuality	DW_AT_visibility
DW_AT_vtable_elem_location	

address — Refers to some location in the address space of the described program.

block — An arbitrary number of uninterpreted bytes of data.

constant — One, two, four or eight bytes of uninterpreted data, or data encoded in the variable length format known as LEB128 (see “Variable Length Data” on page 24-71).

flag — A small constant that indicates the presence or absence of an attribute.

reference — Refers to some member of the set of debugging information entries that describe the program. There are two types of reference. The first is an offset relative to the beginning of the compilation unit in which the reference occurs and must refer to an entry within that same compilation unit. The second type of reference is the address of any debugging information entry within the same executable or shared object; it may refer to an entry in a different compilation unit from the unit containing the reference.

string — A null-terminated sequence of zero or more (non-null) bytes. Data in this form are generally printable strings. Strings may be represented directly in the debugging information entry or as an offset in a separate string table.

There are no limitations on the ordering of attributes within a debugging information entry, but to prevent ambiguity, no more than one attribute with a given name may appear in any debugging information entry.

Relationship of Debugging Information Entries

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

The ownership relation of debugging information entries is achieved naturally because the debugging information is represented as a tree. The nodes of the tree are the debugging information entries themselves. The child entries of any node are exactly those debugging information entries owned by that node.[1]

The tree itself is represented by flattening it in prefix order. Each debugging information entry is defined either to have child entries or not to have child entries (see "Abbreviation Tables" on page 24-67). If an entry is defined not to have children, the next physically succeeding entry is the sibling of the prior entry. If an entry is defined to have children, the next physically succeeding entry is the first child of the prior entry. Additional children of the parent entry are represented as siblings of the first child. A chain of sibling entries is terminated by a null entry.

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

Location Descriptions

The debugging information must provide consumers a way to find the location of program variables, determine the bounds of dynamic arrays and strings and possibly to find the base address of a subroutine's stack frame or the return address of a subroutine. Furthermore, to meet the needs of recent computer architectures and optimization techniques, the debugging information must be able to describe the location of an object whose location changes over the object's lifetime.

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

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

1. Location expressions which are a language independent representation of addressing rules of arbitrary complexity built from a few basic building blocks, or operations. They are sufficient for describing the location of any object as long as its lifetime is either static or the same as the lexical block that owns it, and it does not move throughout its lifetime.

2. Location lists which are used to describe objects that have a limited lifetime or change their location throughout their lifetime. Location lists are more completely described below.

The two forms are distinguished in a context sensitive manner. As the value of an attribute, a location expression is encoded as a block and a location list is encoded as a constant offset into a location list table.

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

Location Expressions

A location expression consists of zero or more location operations. An expression with zero operations is used to denote an object that is present in the source code but not present in the object code (perhaps because of optimization). The location operations fall into two categories, register names and addressing operations. Register names always appear alone and indicate that the referred object is contained inside a particular register. Addressing operations are memory address computation rules. All location operations are encoded as a stream of opcodes that are each followed by zero or more literal operands. The number of operands is determined by the opcode.

Register Name Operators

The following operations can be used to name a register.

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

1. DW_OP_reg0, DW_OP_reg1, ...,DW_OP_reg31
   The DW_OP_regn operations encode the names of up to 32 registers, numbered from 0 through 31, inclusive. The object addressed is in register n.

2. DW_OP_regx
   The DW_OP_regx operation has a single unsigned LEB128 literal operand that encodes the name of a register.

Addressing Operations

Each addressing operation represents a postfix operation on a simple stack machine. Each element of the stack is the size of an address on the target machine. The value on the top of the stack after “executing” the location expression is taken to be the result (the address of the object, or the value of the array bound, or the length of a dynamic string). In the case of

locations used for structure members, the computation assumes that the base address of the containing structure has been pushed on the stack before evaluation of the addressing operation.

Literal Encodings

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

1. DW_OP_lit0, DW_OP_lit1, ...,DW_OP_lit31
   The DW_OP_litn operations encode the unsigned literal values from 0 through 31, inclusive.
2. DW_OP_addr
   The DW_OP_addr operation has a single operand that encodes a machine address and whose size is the size of an address on the target machine.
3. DW_OP_const1u
   The single operand of the DW_OP_const1uoperation provides a 1-byte unsigned integer constant.
4. DW_OP_const1s
   The single operand of the DW_OP_const1soperation provides a 1-byte signed integer constant.
5. DW_OP_const2u
   The single operand of the DW_OP_const2uoperation provides a 2-byte unsigned integer constant.
6. DW_OP_const2s
   The single operand of the DW_OP_const2soperation provides a 2-byte signed integer constant.
7. DW_OP_const4u
   The single operand of the DW_OP_const4uoperation provides a 4-byte unsigned integer constant.
8. DW_OP_const4s
   The single operand of the DW_OP_const4soperation provides a 4-byte signed integer constant.
9. DW_OP_const8u
   The single operand of the DW_OP_const8u operation provides an 8-byte unsigned integer constant.
10. DW_OP_const8s
    The single operand of the DW_OP_const8soperation provides an 8-byte signed integer constant.
11. DW_OP_constu
    The single operand of the DW_OP_constu operation provides an unsigned LEB128 integer constant.
12. DW_OP_consts
    The single operand of the DW_OP_consts operation provides a signed LEB128 integer constant.

Register Based Addressing

The following operations push a value onto the stack that is the result of adding the contents of a register with a given signed offset.

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

2. DW_OP_breg0, DW_OP_breg1, ...,DW_OP_breg31
   The single operand of the DW_OP_bregn operations provides a signed LEB128 offset from the specified register.

3. DW_OP_bregx
   The DW_OP_bregx operation has two operands: a signed LEB128 offset from the specified register which is defined with an unsigned LEB128 number.

Stack Operations

The following operations manipulate the "location stack." Location operations that index the location stack assume that the top of the stack (most recently added entry) has index 0.

1. DW_OP_dup
   The DW_OP_dup operation duplicates the value at the top of the location stack.

2. DW_OP_drop
   The DW_OP_drop operation pops the value at the top of the stack.

3. DW_OP_pick
   The single operand of the DW_OP_pick operation provides a 1-byte index. The stack entry with the specified index (0 through 255, inclusive) is pushed on the stack.

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

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

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

7. DW_OP_deref
   The DW_OP_deref operation pops the top stack entry and treats it as an address. The value retrieved from that address is pushed. The size of the

data retrieved from the dereferenced address is the size of an address on the target machine.

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

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

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

Arithmetic and Logical Operations

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

1. DW_OP_abs
   The DW_OP_abs operation pops the top stack entry and pushes its absolute value.

2. DW_OP_and
   The DW_OP_and operation pops the top two stack values, performs a bitwise and operation on the two, and pushes the result.

3. DW_OP_div
   The DW_OP_div operation pops the top two stack values, divides the

former second entry by the former top of the stack using signed division, and pushes the result.

4. DW_OP_minus
   The DW_OP_minus operation pops the top two stack values, subtracts the former top of the stack from the former second entry, and pushes the result.

5. DW_OP_mod
   The DW_OP_mod operation pops the top two stack values and pushes the result of the calculation: former second stack entry modulo the former top of the stack.

6. DW_OP_mul
   The DW_OP_mul operation pops the top two stack entries, multiplies them together, and pushes the result.

7. DW_OP_neg
   The DW_OP_neg operation pops the top stack entry, and pushes its negation.

8. DW_OP_not
   The DW_OP_not operation pops the top stack entry, and pushes its bitwise complement.

9. DW_OP_or
   The DW_OP_or operation pops the top two stack entries, performs a bitwise or operation on the two, and pushes the result.

10. DW_OP_plus
    The DW_OP_plus operation pops the top two stack entries, adds them together, and pushes the result.

11. DW_OP_plus_uconst
    The DW_OP_plus_uconst operation pops the top stack entry, adds it to the unsigned LEB128 constant operand and pushes the result. This operation is supplied specifically to be able to encode more field offsets in two bytes than can be done with “DW_OP_litn DW_OP_add”.

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

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

14. DW_OP_shra
    The DW_OP_shra operation pops the top two stack entries, shifts the former second entry right (arithmetically) by the number of bits specified by the former top of the stack, and pushes the result.

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

Control Flow Operations

The following operations provide simple control of the flow of a location expression.

1. Relational operators
   The six relational operators each pops the top two stack values, compares the former top of the stack with the former second entry, and pushes the constant value 1 onto the stack if the result of the operation is true or the constant value 0 if the result of the operation is false. The comparisons are done as signed operations. The six operators are DW_OP_le (less than or equal to), DW_OP_ge (greater than or equal to), DW_OP_eq (equal to), DW_OP_lt (less than), DW_OP_gt (greater than) and DW_OP_ne (not equal to).

2. DW_OP_skip
   DW_OP_skip is an unconditional branch. Its single operand is a 2-byte signed integer constant. The 2-byte constant is the number of bytes of the location expression to skip from the current operation, beginning after the 2-byte constant.

3. DW_OP_bra
   DW_OP_bra is a conditional branch. Its single operand is a 2-byte signed integer constant. This operation pops the top of stack. If the value popped is not the constant 0, the 2-byte constant operand is the number of bytes of the location expression to skip from the current operation, beginning after the 2-byte constant.

Special Operations

There are two special operations currently defined:

1. DW_OP_piece
   Many compilers store a single variable in sets of registers, or store a variable partially in memory and partially in registers.DW_OP_pieceprovides a way of describing how large a part of a variable a particular addressing expression refers to.

   DW_OP_piece takes a single argument which is an unsigned LEB128 number. The number describes the size in bytes of the piece of the object referenced by the addressing expression whose result is at the top of the stack.

2. DW_OP_nop
   The DW_OP_nop operation is a place holder. It has no effect on the location stack or any of its values.

Sample Stack Operations

The stack operations defined in “Stack Operations” on page 24-10 are fairly conventional, but the following examples illustrate their behavior graphically.

Before		Operation	After	
0	17	DW_OP_dup	0	17
1	29		1	17
2	1000		2	29
			3	1000
0	17	DW_OP_drop	0	29
1	29		1	1000
2	1000			
0	17	DW_OP_pick 2	0	1000
1	29		1	17
2	1000		2	29
			3	1000
0	17	DW_OP_over	0	29
1	29		1	17
2	1000		2	29
			3	1000
0	17	DW_OP_swap	0	29
1	29		1	17
2	1000		2	1000
0	17	DW_OP_rot	0	29
1	29		1	1000
2	1000		2	17

Example Location Expressions

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

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

DW_OP_reg3
: The value is in register 3.

DW_OP_regx 54
: The value is in register 54.

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

DW_OP_breg11 44

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

DW_OP_fbreg -50

Given an DW_AT_frame_base value of “OPBREG31 64,” this example computes the address of a local variable that is -50 bytes from a logical frame pointer that is computed by adding 64 to the current stack pointer (register 31).

DW_OP_bregx 54 32 DW_OP_deref

A call-by-reference parameter whose address is in the word 32 bytes from where register 54 points.

DW_OP_plus_uconst 4

A structure member is four bytes from the start of the structure instance. The base address is assumed to be already on the stack.

DW_OP_reg3 DW_OP_piece 4 DW_OP_reg10 DW_OP_piece 2

A variable whose first four bytes reside in register 3 and whose next two bytes reside in register 10.

Location Lists

Location lists are used in place of location expressions whenever the object whose location is being described can change location during its lifetime. Location lists are contained in a separate object file section called.debug_loc. A location list is indicated by a location attribute whose value is represented as a constant offset from the beginning of the .debug_loc section to the first byte of the list for the object in question.

Each entry in a location list consists of:

1. A beginning address. This address is relative to the base address of the compilation unit referencing this location list. It marks the beginning of the address range over which the location is valid.
2. An ending address, again relative to the base address of the compilation unit referencing this location list. It marks the first address past the end of the address range over which the location is valid.
3. A location expression describing the location of the object over the range specified by the beginning and end addresses.

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

The end of any given location list is marked by a 0 for the beginning address and a 0 for the end address; no location description is present. A location list containing only such a 0 entry describes an object that exists in the source code but not in the executable program.

Types of Declarations

Any debugging information entry describing a declaration that has a type has a DW_AT_type attribute, whose value is a reference to another debugging information entry. The entry referenced may describe a base type, that is, a type that is not defined in terms of other data types, or it may describe a user-defined type, such as an array, structure or enumeration. Alternatively, the entry referenced may describe a type modifier: constant, packed, pointer, reference or volatile, which in turn will reference another entry describing a type or type modifier (using a DW_AT_type attribute of its own). See "Type Entries" on page 24-34 for descriptions of the entries describing base types, user-defined types and type modifiers.

Accessibility of Declarations

Some languages, notably C++ and Ada, have the concept of the accessibility of an object or of some other program entity. The accessibility specifies which classes of other program objects are permitted access to the object in question.

The accessibility of a declaration is represented by a DW_AT_accessibility attribute, whose value is a constant drawn from the set of codes listed in Table 24-3.

Table 24-3. Accessibility Codes

DW_ACCESS_public
DW_ACCESS_private
DW_ACCESS_protected

Visibility of Declarations

Modula2 has the concept of the visibility of a declaration. The visibility specifies which declarations are to be visible outside of the module in which they are declared.

The visibility of a declaration is represented by a DW_AT_visibility attribute, whose value is a constant drawn from the set of codes listed in Table 24-4.

Table 24-4. Visibility Codes

DW_VIS_local
DW_VIS_exported
DW_VIS_qualified

Virtuality of Declarations

C++ provides for virtual and pure virtual structure or class member functions and for virtual base classes.

The virtuality of a declaration is represented by a DW_AT_virtuality attribute, whose value is a constant drawn from the set of codes listed in Table 24-5.

Table 24-5. Virtuality Codes

DW_VIRTUALITY_none
DW_VIRTUALITY_virtual
DW_VIRTUALITY_pure_virtual

Artificial Entries

A compiler may wish to generate debugging information entries for objects or types that were not actually declared in the source of the application. An example is a formal parameter entry to represent the hidden `this` parameter that most C++ implementations pass as the first argument to non-static member functions.

Any debugging information entry representing the declaration of an object or type artificially generated by a compiler and not explicitly declared by the source program may have a DW_AT_artificial attribute. The value of this attribute is a flag.

Target-Specific Addressing Information

In some systems, addresses are specified as offsets within a given segment rather than as locations within a single flat address space.

Any debugging information entry that contains a description of the location of an object or subroutine may have a DW_AT_segment attribute, whose value is a location description. The description evaluates to the segment value of the item being described. If the entry containing the DW_AT_segment attribute has a DW_AT_low_pc or DW_AT_high_pc attribute, or a location description that evaluates to an address, then those values represent the offset portion of the address within the segment specified by DW_AT_segment.

If an entry has no DW_AT_segment attribute, it inherits the segment value from its parent entry. If none of the entries in the chain of parents for this entry back to its containing compilation unit entry have DW_AT_segment attributes, then the entry is assumed to exist within a flat address space. Similarly, if the entry has a DW_AT_segment attribute containing an empty location description, that entry is assumed to exist within a flat address space.

Some systems support different classes of addresses. The address class may affect the way a pointer is dereferenced or the way a subroutine is called.

Any debugging information entry representing a pointer or reference type or a subroutine or subroutine type may have a DW_AT_address_class attribute, whose value is a constant. The set of permissible values is specific to each target architecture. The value DW_ADDR_none, however, is common to all encodings, and means that no address class has been specified.

For example, the Intel386Sprocessor might use the following values:

Table 24-6. Example Address Class Codes

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

Non-Defining Declarations

A debugging information entry representing a program object or type typically represents the defining declaration of that object or type. In certain contexts, however, a debugger might need information about a declaration of a subroutine, object or type that is not also a definition to evaluate an expression correctly.

As an example, consider the following fragment of C code:

```
void myfunc()
{
    int x;
    {
        extern float x;
        g(x);
    }
}
```

ANSI-C scoping rules require that the value of the variable x passed to the function g is the value of the global variable x rather than of the local version.

Debugging information entries that represent non-defining declarations of a program object or type have a DW_AT_declaration attribute, whose value is a flag.

Declaration Coordinates

It is sometimes useful in a debugger to be able to associate a declaration with its occurrence in the program source.

Any debugging information entry representing the declaration of an object, module, subprogram or type may have DW_AT_decl_file, DW_AT_decl_line and DW_AT_decl_column attributes, each of whose value is a constant.

The value of the DW_AT_decl_file attribute corresponds to a file number from the statement information table for the compilation unit containing this debugging information entry and represents the source file in which the declaration appeared (see “Line Number Information” on page 24-49). The value 0 indicates that no source file has been specified.

The value of the DW_AT_decl_line attribute represents the source line number at which the first character of the identifier of the declared object appears. The value 0 indicates that no source line has been specified.

The value of the DW_AT_decl_column attribute represents the source column number at which the first character of the identifier of the declared object appears. The value 0 indicates that no column has been specified.

Identifier Names

Any debugging information entry representing a program entity that has been given a name may have a DW_AT_name attribute, whose value is a string representing the name as it appears in the source program. A debugging information entry containing no name attribute, or containing a name attribute whose value consists of a name containing a single null byte, represents a program entity for which no name was given in the source.

Note that since the names of program objects described by DWARF are the names as they appear in the source program, implementations of language translators that use some form of mangled name (as do many implementations of C++) should use the unmangled form of the name in the DWARFDW_AT_name attribute, including the keyword `operator`, if present. Sequences of multiple whitespace characters may be compressed.

Program Scope Entries

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

Compilation Unit Entries

An object file may be derived from one or more compilation units. Each such compilation unit will be described by a debugging information entry with the tag DW_TAG_compile_unit.

A compilation unit typically represents the text and data contributed to an executable by a single relocatable object file. It may be derived from several source files, including pre-processed “include files.”

The compilation unit entry may have the following attributes:

1. A DW_AT_low_pc attribute whose value is the relocated address of the first machine instruction generated for that compilation unit.

2. A DW_AT_high_pc attribute whose value is the relocated address of the first location past the last machine instruction generated for that compilation unit.

   The address may be beyond the last valid instruction in the executable, of course, for this and other similar attributes.

   The presence of low and high pc attributes in a compilation unit entry imply that the code generated for that compilation unit is contiguous and exists totally within the boundaries specified by those two attributes. If that is not the case, no low and high pc attributes should be produced.

3. A DW_AT_name attribute whose value is a null-terminated string containing the full or relative path name of the primary source file from which the compilation unit was derived.

4. A DW_AT_language attribute whose constant value is a code indicating the source language of the compilation unit. The set of language names and their meanings are given in Table 24-7.

   Table 24-7. Language Names

DW_LANG_C	Non-ANSI C, such as K&R
DW_LANG_C89	ISO/ANSI C
DW_LANG_C_plus_plus	C++
DW_LANG_Fortran77	FORTRAN77
DW_LANG_Fortran90	Fortran90
DW_LANG_Modula2	Modula2
DW_LANG_Pascal83	ISO/ANSI Pascal

5. A DW_AT_stmt_list attribute whose value is a reference to line number information for this compilation unit.

   This information is placed in a separate object file section from the debugging information entries themselves. The value of the statement list

attribute is the offset in the .debug_line section of the first byte of the line number information for this compilation unit. See “Line Number Information” on page 24-49.

6. A DW_AT_macro_info attribute whose value is a reference to the macro information for this compilation unit.

   This information is placed in a separate object file section from the debugging information entries themselves. The value of the macro information attribute is the offset in the .debug_macinfo section of the first byte of the macro information for this compilation unit. See “Macro Information” on page 24-56.

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

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

8. A DW_AT_producer attribute whose value is a null-terminated string containing information about the compiler that produced the compilation unit. The actual contents of the string will be specific to each producer, but should begin with the name of the compiler vendor or some other identifying character sequence that should avoid confusion with other producer values.

9. A DW_AT_identifier_case attribute whose constant value is a code describing the treatment of identifiers within this compilation unit. The set of identifier case codes is given in Table 24-8.

Table 24-8. Identifier Case Codes

DW_ID_case_sensitive
DW_ID_up_case
DW_ID_down_case
DW_ID_case_insensitive

DW_ID_case_sensitive is the default for all compilation units that do not have this attribute. It indicates that names given as the values of DW_AT_name attributes in debugging information entries for the compilation unit reflect the names as they appear in the source program. The debugger should be sensitive to the case of identifier names when doing identifier lookups.

DW_ID_up_case means that the producer of the debugging information for this compilation unit converted all source names to upper case. The values of the name attributes may not reflect the names as they appear in the source program. The debugger should convert all names to upper case when doing lookups.

DW_ID_down_case means that the producer of the debugging information for this compilation unit converted all source names to lower case. The values of the name attributes may not reflect the names as they appear in the source program. The debugger should convert all names to lower case when doing lookups.

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

10. A DW_AT_base_types attribute whose value is a reference. This attribute points to a debugging information entry representing another compilation unit. It may be used to specify the compilation unit containing the base type entries used by entries in the current compilation unit (see "Base Type Entries" on page 24-34).

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

A compilation unit entry owns debugging information entries that represent the declarations made in the corresponding compilation unit.

Module Entries

Several languages have the concept of a "module."

A module is represented by a debugging information entry with the tag DW_TAG_module. Module entries may own other debugging information entries describing program entities whose declaration scopes end at the end of the module itself.

If the module has a name, the module entry has a DW_AT_name attribute whose value is a null-terminated string containing the module name as it appears in the source program.

If the module contains initialization code, the module entry has a DW_AT_low_pc attribute whose value is the relocated address of the first machine instruction generated for that initialization code. It also has a DW_AT_high_pc attribute whose value is the relocated address of the first location past the last machine instruction generated for the initialization code.

If the module has been assigned a priority, it may have a DW_AT_priority attribute. The value of this attribute is a reference to another debugging information entry describing a variable with a constant value. The value of this variable is the actual constant value of the module's priority, represented as it would be on the target architecture.

A Modula2 definition module may be represented by a module entry containing a DW_AT_declaration attribute.

Subroutine and Entry Point Entries

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

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

General Subroutine and Entry Point Information

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

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

Additional attributes for functions that are members of a class or structure are described in “Structure Member Function Entries” on page 24-41.

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

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

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

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

Subroutine and Entry Point Return Types

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

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

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

A subroutine entry declared with a function prototype style declaration may have a DW_AT_prototyped attribute, whose value is a flag.

Subroutine and Entry Point Locations

A subroutine entry has a DW_AT_low_pc attribute whose value is the relocated address of the first machine instruction generated for the subroutine. It also has a DW_AT_high_pc attribute whose value is the relocated address of the first location past the last machine instruction generated for the subroutine.

Note that for the low and high pc attributes to have meaning, DWARF makes the assumption that the code for a single subroutine is allocated in a single contiguous block of memory.

An entry point has a DW_AT_low_pc attribute whose value is the relocated address of the first machine instruction generated for the entry point.

Subroutines and entry points may also have DW_AT_segmentand DW_AT_address_class attributes, as appropriate, to specify which segments the code for the subroutine resides in and the addressing mode to be used in calling that subroutine.

A subroutine entry representing a subroutine declaration that is not also a definition does not have low and high pc attributes.

Declarations Owned by Subroutines and Entry Points

The declarations enclosed by a subroutine or entry point are represented by debugging information entries that are owned by the subroutine or entry point entry. Entries representing the formal parameters of the subroutine or entry point appear in the same order as the corresponding declarations in the source program.

There is no ordering requirement on entries for declarations that are children of subroutine or entry point entries but that do not represent formal parameters. The formal parameter entries may be interspersed with other entries used by formal parameter entries, such as type entries.

The unspecified parameters of a variable parameter list are represented by a debugging information entry with the tag DW_TAG_unspecified_parameters.

The entry for a subroutine or entry point that includes a Fortran common block has a child entry with the tag DW_TAG_common_inclusion. The common inclusion entry has a DW_AT_common_reference attribute whose value is a reference to the debugging entry for the common block being included (see “Common Block Entries” on page 24-33).

Low-Level Information

A subroutine or entry point entry may have a DW_AT_return_addr attribute, whose value is a location description. The location calculated is the place where the return address for the subroutine or entry point is stored.

A subroutine or entry point entry may also have a DW_AT_frame_base attribute, whose value is a location description that computes the “frame base” for the subroutine or entry point.

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

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

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

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

In the context of supporting nested subroutines, the DW_AT_frame_base attribute value should obey the following constraints:

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

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

Types Thrown by Exceptions

In C++ a subroutine may declare a set of types for which that subroutine may generate or “throw” an exception.

If a subroutine explicitly declares that it may throw an exception for one or more types, each such type is represented by a debugging information entry with the tag DW_TAG_thrown_type. Each such entry is a child of the entry representing the subroutine that may throw this type. All thrown type entries should follow all entries representing the formal parameters of the subroutine and precede all entries representing the local variables or lexical blocks contained in the subroutine. Each thrown type entry contains a DW_AT_type attribute, whose value is a reference to an entry describing the type of the exception that may be thrown.

Function Template Instantiations

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

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

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

Inline Subroutines

A declaration or a definition of an inlinable subroutine is represented by a debugging information entry with the tag DW_TAG_subprogram. The entry for a subroutine that is explicitly declared to be available for inline expansion or that was expanded inline implicitly by the compiler has a DW_AT_inline attribute whose value is a constant. The set of values for the DW_AT_inline attribute is given in Table 24-9.

Table 24-9. Inline Codes

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

Abstract Instances

For the remainder of this discussion, any debugging information entry that is owned (either directly or indirectly) by a debugging information entry that contains the DW_AT_inline attribute will be referred to as an “abstract instance entry.” Any subroutine entry that contains a DW_AT_inline attribute will be known as an “abstract instance root.” Any set of abstract instance entries that are all children (either directly or indirectly) of some abstract instance root, together with the root itself, will be known as an “abstract instance tree.”

A debugging information entry that is a member of an abstract instance tree should not contain a DW_AT_high_pc, DW_AT_low_pc, DW_AT_location, DW_AT_return_addr, DW_AT_start_scope, or DW_AT_segment attribute.

It would not make sense to put these attributes into abstract instance entries since such entries do not represent actual (concrete) instances and thus do not actually exist at run-time.

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

Concrete Inlined Instances

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

Each inlined subroutine entry contains a DW_AT_low_pc attribute, representing the address of the first instruction associated with the given inline expansion. Each inlined subroutine entry also contains a DW_AT_high_pc attribute, representing the address of the first location past the last instruction associated with the inline expansion.

For the remainder of this discussion, any debugging information entry that is owned (either directly or indirectly) by a debugging information entry with the tag DW_TAG_inlined_subroutinewill be referred to as a “concrete inlined instance entry.” Any entry that has the tag DW_TAG_inlined_subroutinewill be known as a “concrete inlined instance root.” Any set of concrete inlined instance entries that are all children (either directly or indirectly) of some concrete inlined instance root, together with the root itself, will be known as a “concrete inlined instance tree.”

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

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

Also, each separate entry within a given concrete inlined instance tree is uniquely associated with one particular entry in the associated abstract instance tree. In other words, there is a one-to-one mapping from entries in a given concrete inlined instance tree to the entries in the associated abstract instance tree.

Note, however, that the reverse is not true. A given abstract instance tree that is associated with a given concrete inlined instance tree may (and quite probably will) contain more entries than the associated concrete inlined instance tree (see below).

Concrete inlined instance entries do not have most of the attributes (except for DW_AT_low_pc, DW_AT_high_pc, DW_AT_location, DW_AT_return_addr, DW_AT_start_scope and DW_AT_segment) that such entries would otherwise normally have. In place of these omitted attributes, each concrete inlined instance entry has a DW_AT_abstract_origin attribute that may be used to obtain the missing information (indirectly) from the associated abstract instance entry. The value of the abstract origin attribute is a reference to the associated abstract instance entry.

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

In general, the structure and content of any given concrete instance tree will be directly analogous to the structure and content of its associated abstract instance tree. There are two exceptions to this general rule however.

1. No entries representing anonymous types are ever made a part of any concrete instance inlined tree.
2. No entries representing members of structure, union or class types are ever made a part of any concrete inlined instance tree.

Entries that represent members and anonymous types are omitted from concrete inlined instance trees because they would simply be redundant duplicates of the corresponding entries in the associated abstract instance trees. If any entry within a concrete inlined instance tree needs to refer to an anonymous type that was declared within the scope of the relevant inline function, the reference should simply refer to the abstract instance entry for the given anonymous type.

If an entry within a concrete inlined instance tree contains attributes describing the declaration coordinates of that entry, then those attributes should refer to the file, line and column of the original declaration of the subroutine, not to the point at which it was inlined.

Out-of-Line Instances of Inline Subroutines

Under some conditions, compilers may need to generate concrete executable instances of inline subroutines other than at points where those subroutines are actually called. For the remainder of this discussion, such concrete instances of inline subroutines will be referred to as “concrete out-of-line instances.”

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

The DWARF representation of a concrete out-of-line instance of an inline subroutine is essentially the same as for a concrete inlined instance of that subroutine (as described in the preceding section). The representation of such a concrete out-of-line instance makes use of DW_AT_abstract_origin attributes in exactly the same way as they are used for a

concrete inlined instance (that is, as references to corresponding entries within the associated abstract instance tree) and, as for concrete instance trees, the entries for anonymous types and for all members are omitted.

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

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

Lexical Block Entries

A lexical block is a bracketed sequence of source statements that may contain any number of declarations. In some languages (C and C++) blocks can be nested within other blocks to any depth.

A lexical block is represented by a debugging information entry with the tag DW_TAG_lexical_block.

The lexical block entry has a DW_AT_low_pc attribute whose value is the relocated address of the first machine instruction generated for the lexical block. The lexical block entry also has a DW_AT_high_pc attribute whose value is the relocated address of the first location past the last machine instruction generated for the lexical block.

If a name has been given to the lexical block in the source program, then the corresponding lexical block entry has a DW_AT_name attribute whose value is a null-terminated string containing the name of the lexical block as it appears in the source program.

This is not the same as a C or C++ label (see below).

The lexical block entry owns debugging information entries that describe the declarations within that lexical block. There is one such debugging information entry for each local declaration of an identifier or inner lexical block.

Label Entries

A label is a way of identifying a source statement. A labeled statement is usually the target of one or more “go to” statements.

A label is represented by a debugging information entry with the tag DW_TAG_label. The entry for a label should be owned by the debugging information entry representing the scope within which the name of the label could be legally referenced within the source program.

The label entry has a DW_AT_low_pc attribute whose value is the relocated address of the first machine instruction generated for the statement identified by the label in the source program. The label entry also has a DW_AT_name attribute whose value is a null-terminated string containing the name of the label as it appears in the source program.

With Statement Entries

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

A with statement is represented by a debugging information entry with the tag DW_TAG_with_stmt. A with statement entry has a DW_AT_low_pc attribute whose value is the relocated address of the first machine instruction generated for the body of the with statement. A with statement entry also has a DW_AT_high_pc attribute whose value is the relocated address of the first location after the last machine instruction generated for the body of the statement.

The with statement entry has a DW_AT_type attribute, denoting the type of record whose fields may be referenced without full qualification within the body of the statement. It also has a DW_AT_location attribute, describing how to find the base address of the record object referenced within the body of the with statement.

Try and Catch Block Entries

In C++ a lexical block may be designated as a "catch block." A catch block is an exception handler that handles exceptions thrown by an immediately preceding "try block." A catch block designates the type of the exception that it can handle.

A try block is represented by a debugging information entry with the tag DW_TAG_try_block. A catch block is represented by a debugging information entry with the tag DW_TAG_catch_block. Both try and catch block entries contain a DW_AT_low_pc attribute whose value is the relocated address of the first machine instruction generated for that block. These entries also contain a DW_AT_high_pc attribute whose value is the relocated address of the first location past the last machine instruction generated for that block.

Catch block entries have at least one child entry, an entry representing the type of exception accepted by that catch block. This child entry will have one of the tags DW_TAG_formal_parameter or DW_TAG_unspecified_parameters, and will have the same form as other parameter entries.

The first sibling of each try block entry will be a catch block entry.

Data Object and Object List Entries

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

Data Object Entries

Program variables, formal parameters and constants are represented by debugging information entries with the tags DW_TAG_variable, DW_TAG_formal_parameter and DW_TAG_constant, respectively.

The tag DW_TAG_constant is used for languages that distinguish between variables that may have constant value and true named constants.

The debugging information entry for a program variable, formal parameter or constant may have the following attributes:

1. A DW_AT_name attribute whose value is a null-terminated string containing the data object name as it appears in the source program.

   If a variable entry describes a C++ anonymous union, the name attribute is omitted or consists of a single zero byte.

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

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

3. A DW_AT_location attribute, whose value describes the location of a variable or parameter at run-time.

   A data object entry representing a non-defining declaration of the object will not have a location attribute, and will have the DW_AT_declaration attribute.

   In a variable entry representing the definition of the variable (that is, with no DW_AT_declaration attribute) if no location attribute is present, or if the location attribute is present but describes a null entry (as described in “Location Descriptions” on page 24-7), the variable is assumed to exist in the source code but not in the executable program (but see number 9, below).

   The location of a variable may be further specified with a DW_AT_segment attribute, if appropriate.

4. A DW_AT_type attribute describing the type of the variable, constant or formal parameter.

5. If the variable entry represents the defining declaration for a C++ static data member of a structure, class or union, the entry has a DW_AT_specification attribute, whose value is a reference to the debugging information entry representing the declaration of this data member. The referenced entry will be a child of some class, structure or union type entry.

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

6. Some languages distinguish between parameters whose value in the calling function can be modified by the callee (variable parameters), and parameters whose value in the calling function cannot be modified by the callee (constant parameters).

   If a formal parameter entry represents a parameter whose value in the calling function may be modified by the callee, that entry may have a DW_AT_variable_parameter attribute, whose value is a flag. The absence of this attribute implies that the parameter's value in the calling function cannot be modified by the callee.

7. Fortran90 has the concept of an optional parameter.

   If a parameter entry represents an optional parameter, it has a DW_AT_is_optional attribute, whose value is a flag.

8. A formal parameter entry describing a formal parameter that has a default value may have a DW_AT_default_value attribute. The value of this attribute is a reference to the debugging information entry for a variable or subroutine. The default value of the parameter is the value of the variable (which may be constant) or the value returned by the subroutine. If the value of the DW_AT_default_value attribute is 0, it means that no default value has been specified.

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

10. If the scope of an object begins sometime after the low pc value for the scope most closely enclosing the object, the object entry may have a DW_AT_start_scope attribute. The value of this attribute is the offset in bytes of the beginning of the scope for the object from the low pc value of the debugging information entry that defines its scope.

    The scope of a variable may begin somewhere in the middle of a lexical block in a language that allows executable code in a block before a variable declaration, or where one declaration containing initialization code may

change the scope of a subsequent declaration. For example, in the following C code:

```
float x = 99.99;

int myfunc()
{
    float f = x;
    float x = 88.99;

    return 0;
}
```

ANSI-C scoping rules require that the value of the variable x assigned to the variable f in the initialization sequence is the value of the global variable x, rather than the local x, because the scope of the local variable x only starts after the full declarator for the local x.

Common Block Entries

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

Imported Declaration Entries

Some languages support the concept of importing into a given module declarations made in a different module.

An imported declaration is represented by a debugging information entry with the tag DW_TAG_imported_declaration. The entry for the imported declaration has a DW_AT_name attribute whose value is a null-terminated string containing the name of the entity whose declaration is being imported as it appears in the source program. The imported declaration entry also has a DW_AT_import attribute, whose value is a reference to the debugging information entry representing the declaration that is being imported.

Namelist Entries

At least one language, Fortran90, has the concept of a namelist. A namelist is an ordered list of the names of some set of declared objects. The namelist object itself may be used as a replacement for the list of names in various contexts.

A namelist is represented by a debugging information entry with the tag DW_TAG_namelist.

If the namelist itself has a name, the namelist entry has a DW_AT_name attribute, whose value is a null-terminated string containing the namelist's name as it appears in the source program.

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

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

Type Entries

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

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

Base Type Entries

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

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

A base type entry also has a DW_AT_encoding attribute describing how the base type is encoded and is to be interpreted. The value of this attribute is a constant. The set of values and their meanings for the DW_AT_encoding attribute is given in Table 24-10.

Table 24-10. Encoding Attribute Values

Name	Meaning
DW_ATE_address	linear machine address
DW_ATE_boolean	true or false
DW_ATE_complex_float	complex floating-point number
DW_ATE_float	floating-point number

Table 24-10. Encoding Attribute Values (Cont.)

Name	Meaning
DW_ATE_signed	signed binary integer
DW_ATE_signed_char	signed character
DW_ATE_unsigned	unsigned binary integer
DW_ATE_unsigned_char	unsigned character

All encodings assume the representation that is “normal” for the target architecture.

A base type entry has a DW_AT_byte_size attribute, whose value is a constant, describing the size in bytes of the storage unit used to represent an object of the given type.

If the value of an object of the given type does not fully occupy the storage unit described by the byte size attribute, the base type entry may have a DW_AT_bit_size attribute and a DW_AT_bit_offset attribute, both of whose values are constants. The bit size attribute describes the actual size in bits used to represent a value of the given type. The bit offset attribute describes the offset in bits of the high order bit of a value of the given type from the high order bit of the storage unit used to contain that value.

For example, the C type `int` on a machine that uses 32-bit integers would be represented by a base type entry with a name attribute whose value was “int,” an encoding attribute whose value was DW_ATE_signed and a byte size attribute whose value was4.

Type Modifier Entries

A base or user-defined type may be modified in different ways in different languages. A type modifier is represented in DWARF by a debugging information entry with one of the tags given in Table 24-11.

Table 24-11. Type Modifier Tags

Tag	Meaning
DW_TAG_const_type	C or C++ const qualified type
DW_TAG_packed_type	Pascal packed type
DW_TAG_pointer_type	The address of the object whose type is being modified
DW_TAG_reference_type	A C++ reference to the object whose type is being modified
DW_TAG_volatile_type	C or C++ volatile qualified type

Each of the type modifier entries has a DW_AT_type attribute, whose value is a reference to a debugging information entry describing a base type, a user-defined type or another type modifier.

A modified type entry describing a pointer or reference type may have a DW_AT_address_class attribute to describe how objects having the given pointer or reference type ought to be dereferenced.

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

As examples of how type modifiers are ordered, take the following C declarations:

```
const char * volatile p;
```

which represents a volatile pointer to a constant character. This is encoded in DWARF as:

```
DW_TAG_volatile_type -->
    DW_TAG_pointer_type -->
        DW_TAG_const_type -->
            DW_TAG_base_type
```

```
volatile char * const p;
```

on the other hand, represents a constant pointer to a volatile character. This is encoded as:

```
DW_TAG_const_type -->
    DW_TAG_pointer_type -->
        DW_TAG_volatile_type -->
            DW_TAG_base_type
```

Typedef Entries

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

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

Array Type Entries

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

An array type is represented by a debugging information entry with the tag DW_TAG_array_type.

If a name has been given to the array type in the source program, then the corresponding array type entry has a DW_AT_name attribute whose value is a null-terminated string containing the array type name as it appears in the source program.

The array type entry describing a multidimensional array may have a DW_AT_ordering attribute whose constant value is interpreted to mean either row-major or column-major ordering of array elements. The required attribute names are listed in Table 24-12. If no ordering attribute is present, the default ordering for the source language (which is indicated by the DW_AT_language attribute of the enclosing compilation unit entry) is assumed.

Table 24-12. Array Ordering

DW_ORD_col_major
DW_ORD_row_major

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

An array type entry has a DW_AT_type attribute describing the type of each element of the array.

If the amount of storage allocated to hold each element of an object of the given array type is different from the amount of storage that is normally allocated to hold an individual object of the indicated element type, then the array type entry has a DW_AT_stride_size attribute, whose constant value represents the size in bits of each element of the array.

If the size of the entire array can be determined statically at compile time, the array type entry may have a DW_AT_byte_size attribute, whose constant value represents the total size in bytes of an instance of the array type.

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

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

In languages, such as ANSI-C, in which there is no concept of a “multidimensional array,” an array of arrays may be represented by a debugging information entry for a multidimensional array.

Structure, Union, and Class Type Entries

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

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

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

Among the languages discussed in this document, the “class” concept is unique to C++. A class is similar to a structure. A C++ class or structure may have “member functions” which are subroutines that are within the scope of a class or structure.

General Structure Description

Structure, union, and class types are represented by debugging information entries with the tags DW_TAG_structure_type, DW_TAG_union_type and DW_TAG_class_type, respectively. If a name has been given to the structure, union, or class in the source program, then the corresponding structure type, union type, or class type entry has a DW_AT_name attribute whose value is a null-terminated string containing the type name as it appears in the source program.

If the size of an instance of the structure type, union type, or class type entry can be determined statically at compile time, the entry has a DW_AT_byte_size attribute whose constant value is the number of bytes required to hold an instance of the structure, union, or class, and any padding bytes.

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

The members of a structure, union, or class are represented by debugging information entries that are owned by the corresponding structure type, union type, or class type entry and appear in the same order as the corresponding declarations in the source program.

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

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

Derived Classes and Structures

The class type or structure type entry that describes a derived class or structure owns debugging information entries describing each of the classes or structures it is derived from, ordered as they were in the source program. Each such entry has the tag DW_TAG_inheritance.

An inheritance entry has a DW_AT_type attribute whose value is a reference to the debugging information entry describing the structure or class from which the parent structure or class of the inheritance entry is derived. It also has a DW_AT_data_member_location attribute, whose value is a location description describing the location of the beginning of the data members contributed to the entire class by this subobject relative to the beginning address of the data members of the entire class.

An inheritance entry may have a DW_AT_accessibility attribute. If no accessibility attribute is present, private access is assumed. If the structure or class referenced by the inheritance entry serves as a virtual base class, the inheritance entry has a DW_AT_virtuality attribute.

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

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

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

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

Friends

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

A friend entry has a DW_AT_friend attribute, whose value is a reference to the debugging information entry describing the declaration of the friend.

Structure Data Member Entries

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

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

If the member entry is defined in the structure or class body, it has a DW_AT_data_member_location attribute whose value is a location description that describes the location of that member relative to the base address of the structure, union, or class that most closely encloses the corresponding member declaration.

The addressing expression represented by the location description for a structure data member expects the base address of the structure data member to be on the expression stack before being evaluated.

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

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

1. A DW_AT_byte_size attribute whose constant value is the number of bytes that contain an instance of the bit field and any padding bits.

   The byte size attribute may be omitted if the size of the object containing the bit field can be inferred from the type attribute of the data member containing the bit field.

2. A DW_AT_bit_offset attribute whose constant value is the number of bits to the left of the leftmost (most significant) bit of the bit field value.

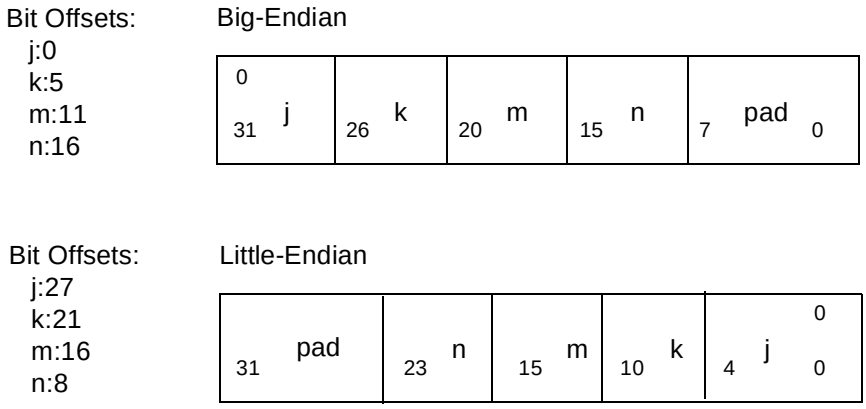
3. A DW_AT_bit_size attribute whose constant value is the number of bits occupied by the bit field value.

The location description for a bit field calculates the address of an anonymous object containing the bit field. The address is relative to the structure, union, or class that most closely encloses the bit field declaration. The number of bytes in this anonymous object is the value of the byte size attribute of the bit field. The offset (in bits) from the most significant bit of the anonymous object to the most significant bit of the bit field is the value of the bit offset attribute.

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

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

In both cases, the location descriptions for the debugging information entries for j, k, m and n describe the address of the same 32-bit word that contains all three members. (In the big-endian case, the location description addresses the most significant byte, in the little-endian case, the least significant). The following diagram shows the structure layout and lists the bit offsets for each case. The offsets are from the most significant bit of the object addressed by the location description.



Structure Member Function Entries

A member function is represented in the debugging information by a debugging information entry with the tag DW_TAG_subprogram. The member function entry may contain the same attributes and follows the same rules as non-member global subroutine entries (see “Subroutine and Entry Point Entries” on page 24-23).

If the member function entry describes a virtual function, then that entry has a DW_AT_virtuality attribute.

An entry for a virtual function also has a DW_AT_vtable_elem_location attribute whose value contains a location description yielding the address of the slot for the function within the virtual function table for the enclosing class or structure.

If the member function entry represents the defining declaration of a member function and that definition appears outside of the body of the enclosing class or structure declaration, the member function entry has a DW_AT_specification attribute, whose value is a reference to the debugging information entry representing the declaration of this function member. The referenced entry will be a child of some class or structure type entry.

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

Class Template Instantiations

In C++ a class template is a generic definition of a class type that is instantiated differently when an instance of the class is declared or defined. The generic description of the class may include both parameterized types and parameterized constant values. DWARF does not represent the generic template definition, but does represent each instantiation.

A class template instantiation is represented by a debugging information with the tag DW_TAG_class_type. With four exceptions, such an entry will contain the same attributes

and have the same types of child entries as would an entry for a class type defined explicitly using the instantiation types and values. The exceptions are:

1. Each formal parameterized type declaration appearing in the template definition is represented by a debugging information entry with the tag DW_TAG_template_type_parameter. Each such entry has a DW_AT_name attribute, whose value is a null-terminated string containing the name of the formal type parameter as it appears in the source program. The template type parameter entry also has a DW_AT_type attribute describing the actual type by which the formal is replaced for this instantiation.

2. Each formal parameterized value declaration appearing in the templated definition is represented by a debugging information entry with the tag DW_TAG_template_value_parameter. Each such entry has a DW_AT_name attribute, whose value is a null-terminated string containing the name of the formal value parameter as it appears in the source program. The template value parameter entry also has a DW_AT_type attribute describing the type of the parameterized value. Finally, the template value parameter entry has a DW_AT_const_value attribute, whose value is the actual constant value of the value parameter for this instantiation as represented on the target architecture.

3. If the compiler has generated a special compilation unit to hold the template instantiation and that compilation unit has a different name from the compilation unit containing the template definition, the name attribute for the debugging entry representing that compilation unit should be empty or omitted.

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

Variant Entries

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

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

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

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

Alternatively, the variant entry may contain a DW_AT_discr_list attribute, whose value represents a list of discriminant values. This list is represented by any of the block forms and may contain a mixture of case labels and label ranges. Each item on the list is prefixed with a discriminant value descriptor that determines whether the list item represents a single label or a label range. A single case label is represented as an LEB128 number as defined above for the DW_AT_discr_value attribute. A label range is represented by two LEB128 numbers, the low value of the range followed by the high value. Both values follow the rules for signedness just described. The discriminant value descriptor is a constant that may have one of the values given in Table 24-13.

Table 24-13. Discriminant Descriptor Values

DW_DSC_label
DW_DSC_range

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

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

Enumeration Type Entries

An “enumeration type” is a scalar that can assume one of a fixed number of symbolic values.

An enumeration type is represented by a debugging information entry with the tag DW_TAG_enumeration_type.

If a name has been given to the enumeration type in the source program, then the corresponding enumeration type entry has a DW_AT_name attribute whose value is a null-terminated string containing the enumeration type name as it appears in the source program. These entries also have a DW_AT_byte_size attribute whose constant value is the number of bytes required to hold an instance of the enumeration.

Each enumeration literal is represented by a debugging information entry with the tag DW_TAG_enumerator. Each such entry is a child of the enumeration type entry, and the enumerator entries appear in the same order as the declarations of the enumeration literals in the source program.

Each enumerator entry has a DW_AT_name attribute, whose value is a null-terminated string containing the name of the enumeration literal as it appears in the source program. Each enumerator entry also has a DW_AT_const_value attribute, whose value is the actual numeric value of the enumerator as represented on the target system.

Subroutine Type Entries

It is possible in C to declare pointers to subroutines that return a value of a specific type. In both ANSI C and C++, it is possible to declare pointers to subroutines that not only return a value of a specific type, but accept only arguments of specific types. The type of such pointers would be described with a “pointer to” modifier applied to a user-defined type.

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

If the subroutine type describes a function that returns a value, then the subroutine type entry has a DW_AT_type attribute to denote the type returned by the subroutine. If the types of the arguments are necessary to describe the subroutine type, then the corresponding subroutine type entry owns debugging information entries that describe the arguments. These debugging information entries appear in the order that the corresponding argument types appear in the source program.

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

A subroutine entry declared with a function prototype style declaration may have a DW_AT_prototyped attribute, whose value is a flag.

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

1. Each debugging information entry that is owned by a subroutine type entry and that defines a single argument of a specific type has the tag DW_TAG_formal_parameter.

   The formal parameter entry has a type attribute to denote the type of the corresponding formal parameter.

2. The unspecified parameters of a variable parameter list are represented by a debugging information entry owned by the subroutine type entry with the tag DW_TAG_unspecified_parameters.

String Type Entries

A “string” is a sequence of characters that have specific semantics and operations that separate them from arrays of characters. Fortran is one of the languages that has a string type.

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

The string type entry may have a DW_AT_string_length attribute whose value is a location description yielding the location where the length of the string is stored in the pro-

gram. The string type entry may also have a DW_AT_byte_size attribute, whose constant value is the size in bytes of the data to be retrieved from the location referenced by the string length attribute. If no byte size attribute is present, the size of the data to be retrieved is the same as the size of an address on the target machine.

If no string length attribute is present, the string type entry may have a DW_AT_byte_size attribute, whose constant value is the length in bytes of the string.

Set Entries

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

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

The set type entry has a DW_AT_type attribute to denote the type of an element of the set.

If the amount of storage allocated to hold each element of an object of the given set type is different from the amount of storage that is normally allocated to hold an individual object of the indicated element type, then the set type entry has a DW_AT_byte_size attribute, whose constant value represents the size in bytes of an instance of the set type.

Subrange Type Entries

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

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

The subrange entry may have a DW_AT_type attribute to describe the type of object of whose values this subrange is a subset.

If the amount of storage allocated to hold each element of an object of the given subrange type is different from the amount of storage that is normally allocated to hold an individual object of the indicated element type, then the subrange type entry has a DW_AT_byte_size attribute, whose constant value represents the size in bytes of each element of the subrange type.

The subrange entry may have the attributes DW_AT_lower_bound and DW_AT_upper_bound to describe, respectively, the lower and upper bound values of the subrange. The DW_AT_upper_bound attribute may be replaced by a DW_AT_count attribute, whose value describes the number of elements in the subrange rather than the value of the last element. If a bound or count value is described by a constant not represented in the program's address space and can be represented by one of the constant

attribute forms, then the value of the lower or upper bound or count attribute may be one of the constant types. Otherwise, the value of the lower or upper bound or count attribute is a reference to a debugging information entry describing an object containing the bound value or itself describing a constant value.

If either the lower or upper bound or count values are missing, the bound value is assumed to be a language-dependent default constant.

The default lower bound value for C or C++ is 0. For Fortran, it is 1. No other default values are currently defined by DWARF.

If the subrange entry has no type attribute describing the basis type, the basis type is assumed to be the same as the object described by the lower bound attribute (if it references an object). If there is no lower bound attribute, or it does not reference an object, the basis type is the type of the upper bound or count attribute (if it references an object). If there is no upper bound or count attribute or it does not reference an object, the type is assumed to be the same type, in the source language of the compilation unit containing the subrange entry, as a signed integer with the same size as an address on the target machine.

Pointer to Member Type Entries

In C++, a pointer to a data or function member of a class or structure is a unique type.

A debugging information entry representing the type of an object that is a pointer to a structure or class member has the tag DW_TAG_ptr_to_member_type.

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

The pointer to member entry has a DW_AT_type attribute to describe the type of the class or structure member to which objects of this type may point. The pointer to member entry also has a DW_AT_containing_type attribute, whose value is a reference to a debugging information entry for the class or structure to whose members objects of this type may point.

Finally, the pointer to member entry has a DW_AT_use_location attribute whose value is a location description that computes the address of the member of the class or structure to which the pointer to member type entry can point.

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

The DW_AT_use_locationexpression, however, cannot be used on its own, but must be used in conjunction with the location expressions for a particular object of the given pointer to member type and for a particular structure or class instance. The DW_AT_use_location attribute expects two values to be pushed onto the location expression stack before the DW_AT_use_locationexpression is evaluated. The first value pushed should be the value of the pointer to member object itself. The second value pushed should be the base address of the entire structure or union instance containing the member whose address is being calculated.

So, for an expression like

```
object.*mbr_ptr
```

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

1. Push the value of mbr_ptr onto the location expression stack.
2. Push the base address of object onto the location expression stack.
3. Evaluate the DW_AT_use_locationexpression for the type of mbr_ptr.

File Type Entries

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

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

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

The file type entry also has a DW_AT_byte_size attribute, whose value is a constant representing the size in bytes of an instance of this file type.

Other Debugging Information

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

Accelerated Access

A debugger frequently needs to find the debugging information for a program object defined outside of the compilation unit where the debugged program is currently stopped. Sometimes it will know only the name of the object; sometimes only the address. To find the debugging information associated with a global object by name, using the DWARF debugging information entries alone, a debugger would need to run through all entries at the highest scope within each compilation unit. For lookup by address, for a subroutine, a debugger can use the low and high pc attributes of the compilation unit entries to quickly narrow down the search, but these attributes only cover the range of addresses for the text associated with a compilation unit entry.

To find the debugging information associated with a data object, an exhaustive search would be needed. Furthermore, any search through debugging information entries for dif-

ferent compilation units within a large program would potentially require the access of many memory pages, probably hurting debugger performance.

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

Lookup by Name

For lookup by name, a table is maintained in a separate object file section called .debug_pubnames. The table consists of sets of variable length entries, each set describing the names of global objects whose definitions or declarations are represented by debugging information entries owned by a single compilation unit. Each set begins with a header containing four values: the total length of the entries for that set, not including the length field itself, a version number, the offset from the beginning of the .debug_info section of the compilation unit entry referenced by the set and the size in bytes of the contents of the .debug_info section generated to represent that compilation unit. This header is followed by a variable number of offset/name pairs. Each pair consists of the offset from the beginning of the compilation unit entry corresponding to the current set to the debugging information entry for the given object, followed by a null-terminated character string representing the name of the object as given by the DW_AT_name attribute of the referenced debugging entry. Each set of names is terminated by zero.

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

Lookup by Address

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

1. The total length of the entries for that set, not including the length field itself.
2. A version number.
3. The offset from the beginning of the .debug_info section of the compilation unit entry referenced by the set.
4. The size in bytes of an address on the target architecture. For segmented addressing, this is the size of the offset portion of the address.
5. The size in bytes of a segment descriptor on the target architecture. If the target system uses a flat address space, this value is 0.

This header is followed by a variable number of address range descriptors. Each descriptor is a pair consisting of the beginning address of a range of text or data covered by some entry owned by the corresponding compilation unit entry, followed by the length of that

range. A particular set is terminated by an entry consisting of two zeroes. By scanning the table, a debugger can quickly decide which compilation unit to look in to find the debugging information for an object that has a given address.

Line Number Information

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

As mentioned in “Compilation Unit Entries” on page 24-20, above, the line number information generated for a compilation unit is represented in the .debug_line section of an object file and is referenced by a corresponding compilation unit debugging information entry in the .debug_info section.

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

- the source file name
- the source line number
- the source column number
- whether this instruction is the beginning of a source statement
- whether this instruction is the beginning of a basic block.

Such a matrix, however, would be impractically large. We shrink it with two techniques. First, we delete from the matrix each row whose file, line and source column information is identical with that of its predecessors. Second, we design a byte-coded language for a state machine and store a stream of bytes in the object file instead of the matrix. This language can be much more compact than the matrix. When a consumer of the statement information executes, it must “run” the state machine to generate the matrix for each compilation unit it is interested in. The concept of an encoded matrix also leaves room for expansion. In the future, columns can be added to the matrix to encode other things that are related to individual instruction addresses.

Definitions

The following terms are used in the description of the line number information format:

state machine	The hypothetical machine used by a consumer of the line number information to expand the byte-coded instruction stream into a matrix of line number information.
statement program	A series of byte-coded line number information instructions representing one compilation unit.

basic block	A sequence of instructions that is entered only at the first instruction and exited only at the last instruction. We define a procedure invocation to be an exit from a basic block.
sequence	A series of contiguous target machine instructions. One compilation unit may emit multiple sequences (that is, not all instructions within a compilation unit are assumed to be contiguous).
sbyte	Small signed integer.
ubyte	Small unsigned integer.
uhalf	Medium unsigned integer.
sword	Large signed integer.
uword	Large unsigned integer.
LEB128	Variable length signed and unsigned data. See “Variable Length Data” on page 24-71.

State Machine Registers

The statement information state machine has the following registers:

address	The program-counter value corresponding to a machine instruction generated by the compiler.
file	An unsigned integer indicating the identity of the source file corresponding to a machine instruction.
line	An unsigned integer indicating a source line number. Lines are numbered beginning at 1. The compiler may emit the value 0 in cases where an instruction cannot be attributed to any source line.
column	An unsigned integer indicating a column number within a source line. Columns are numbered beginning at 1. The value 0 is reserved to indicate that a statement begins at the “left edge” of the line.
is_stmt	A boolean indicating that the current instruction is the beginning of a statement.
basic_block	A boolean indicating that the current instruction is the beginning of a basic block.
end_sequence	A boolean indicating that the current address is that of the first byte after the end of a sequence of target machine instructions.

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

```
address         0
file            1
line            1
column          0
```

```
is_stmt          determined by default_is_stmt in the
                 statement program prologue
basic_block      "false"
end_sequence     "false"
```

Statement Program Instructions

The state machine instructions in a statement program belong to one of three categories:

special opcodes — These have a ubyte opcode field and no arguments. Most of the instructions in a statement program are special opcodes.

standard opcodes — These have a ubyte opcode field which may be followed by zero or more LEB128 arguments (except for DW_LNS_fixed_advance_pc, see below). The opcode implies the number of arguments and their meanings, but the statement program prologue also specifies the number of arguments for each standard opcode.

extended opcodes — These have a multiple byte format. The first byte is zero; the next bytes are an unsigned LEB128 integer giving the number of bytes in the instruction itself (does not include the first zero byte or the size). The remaining bytes are the instruction itself.

The Statement Program Prologue

The optimal encoding of line number information depends to a certain degree upon the architecture of the target machine. The statement program prologue provides information used by consumers in decoding the statement program instructions for a particular compilation unit and also provides information used throughout the rest of the statement program. The statement program for each compilation unit begins with a prologue containing the following fields in order:

1. total_length(uword)
   The size in bytes of the statement information for this compilation unit (not including the total_length field itself).
2. version(uhalf)
   Version identifier for the statement information format.
3. prologue_length(uword)
   The number of bytes following the prologue_length field to the beginning of the first byte of the statement program itself.
4. minimum_instruction_length(ubyte)
   The size in bytes of the smallest target machine instruction. Statement program opcodes that alter the address register first multiply their operands by this value.
5. default_is_stmt(ubyte)
   The initial value of the is_stmt register.

A simple code generator that emits machine instructions in the order implied by the source program would set this to “true,” and every entry in the matrix would represent a statement

boundary. A pipeline scheduling code generator would set this to “false” and emit a specific statement program opcode for each instruction that represented a statement boundary.

6. line_base(sbyte)
   This parameter affects the meaning of the special opcodes. See below.

7. line_range(ubyte)
   This parameter affects the meaning of the special opcodes. See below.

8. opcode_base(ubyte)
   The number assigned to the first special opcode.

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

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

11. file_names(sequence of file entries)
    The sequence contains an entry for each source file that contributed to the statement information for this compilation unit or is used in other contexts, such as in a declaration coordinate or a macro file inclusion. Each entry has a null-terminated string containing the file name, an unsigned LEB128 number representing the directory index of the directory in which the file was found, an unsigned LEB128 number representing the time of last modification for the file and an unsigned LEB128 number representing the length in bytes of the file. A compiler may choose to emit LEB128(0) for the time and length fields to indicate that this information is not available. The last entry is followed by a single null byte.

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

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

    A compiler may generate a single null byte for the file names field and define file names using the extended opcode DEFINE_FILE.

The Statement Program

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

Special Opcodes

Each 1-byte special opcode has the following effect on the state machine:

1. Add a signed integer to the line register.
2. Multiply an unsigned integer by theminimum_instruction_length field of the statement program prologue and add the result to the address register.
3. Append a row to the matrix using the current values of the state machine registers.
4. Set the basic_block register to “false.”

All of the special opcodes do those same four things; they differ from one another only in what values they add to thelineandaddressregisters.

Instead of assigning a fixed meaning to each special opcode, the statement program uses several parameters in the prologue to configure the instruction set. There are two reasons for this. First, although the opcode space available for special opcodes now ranges from 10 through 255, the lower bound may increase if one adds new standard opcodes. Thus, the opcode_base field of the statement program prologue gives the value of the first special opcode. Second, the best choice of special-opcode meanings depends on the target architecture. For example, for a RISC machine where the compiler-generated code interleaves instructions from different lines to schedule the pipeline, it is important to be able to add a negative value to the line register to express the fact that a later instruction may have been emitted for an earlier source line. For a machine where pipeline scheduling never occurs, it is advantageous to trade away the ability to decrease the line register (a standard opcode provides an alternate way to decrease the line number) in return for the ability to add larger positive values to the address register. To permit this variety of strategies, the statement program prologue defines a line_base field that specifies the minimum value which a special opcode can add to the line register and a line_range field that defines the range of values it can add to the line register.

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

```
opcode = (desired line increment - line_base) +
(line_range * address advance) + opcode_base
```

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

To decode a special opcode, subtract the opcode_base from the opcode itself. The amount to increment the address register is the adjusted opcode divided by the line_range. The amount to increment the line register is the line_base plus the result of the adjusted opcode modulo the line_range. That is,

```
line increment = line_base + (adjusted opcode % line_range)
```

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

The opcode mapping would be:

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

There is no requirement that the expression255 - line_base + 1be an integral multiple ofline_range.

Standard Opcodes

There are currently 9 standard ubyte opcodes. In the future additional ubyte opcodes may be defined by setting the opcode_base field in the statement program prologue to a value greater than 10.

1. DW_LNS_copy
   Takes no arguments.Append a row to the matrix using the current values of the state-machine registers. Then set the basic_block register to “false.”

2. DW_LNS_advance_pc
   Takes a single unsigned LEB128 operand, multiplies it by the minimum_instruction_length field of the prologue, and adds the result to the address register of the state machine.

3. DW_LNS_advance_line
   Takes a single signed LEB128 operand and adds that value to the line register of the state machine.

4. DW_LNS_set_file
   Takes a single unsigned LEB128 operand and stores it in the file register of the state machine.

5. DW_LNS_set_column
   Takes a single unsigned LEB128 operand and stores it in the column register of the state machine.

6. DW_LNS_negate_stmt
   Takes no arguments.Set the is_stmt register of the state machine to the logical negation of its current value.

7. DW_LNS_set_basic_block
   Takes no arguments.Set the basic_block register of the state machine to "true."

8. DW_LNS_const_add_pc
   Takes no arguments.Add to the address register of the state machine the address increment value corresponding to special opcode 255.

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

9. DW_LNS_fixed_advance_pc
   Takes a single uhalf operand. Add to the address register of the state machine the value of the (unencoded) operand. This is the only extended opcode that takes an argument that is not a variable length number.

   The motivation for DW_LNS_fixed_advance_pc is this: existing assemblers cannot emit DW_LNS_advance_pc or special opcodes because they cannot encode LEB128 numbers or judge when the computation of a special opcode overflows and requires the use of DW_LNS_advance_pc. Such assemblers, however, can use DW_LNS_fixed_advance_pcinstead, sacrificing compression.

Extended Opcodes

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

1. DW_LNE_end_sequence
   Set the end_sequence register of the state machine to "true" and append a row to the matrix using the current values of the state-machine registers. Then reset the registers to the initial values specified above.

   Every statement program sequence must end with a

DW_LNE_end_sequence instruction which creates a row whose address is that of the byte after the last target machine instruction of the sequence.

2. DW_LNE_set_address
   Takes a single relocatable address as an operand. The size of the operand is the size appropriate to hold an address on the target machine. Set the address register to the value given by the relocatable address.

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

3. DW_LNE_define_file
   Takes 4 arguments. The first is a null terminated string containing a source file name. The second is an unsigned LEB128 number representing the directory index of the directory in which the file was found. The third is an unsigned LEB128 number representing the time of last modification of the file. The fourth is an unsigned LEB128 number representing the length in bytes of the file. The time and length fields may contain LEB128(0) if the information is not available.

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

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

"Appendix 3 -- Statement Program Examples" on page 24-99 gives some sample statement programs.

Macro Information

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

As described in "Compilation Unit Entries" on page 24-20, the macro information for a given compilation unit is represented in the .debug_macinfo section of an object file. The macro information for each compilation unit is represented as a series of "macinfo" entries. Each macinfo entry consists of a "type code" and up to two additional operands. The series of entries for a given compilation unit ends with an entry containing a type code of 0.

Macinfo Types

The valid macinfo types are as follows:

DW_MACINFO_define	A macro definition.
DW_MACINFO_undef	A macro un-definition.
DW_MACINFO_start_file	The start of a new source file inclusion.
DW_MACINFO_end_file	The end of the current source file inclusion.
DW_MACINFO_vendor_ext	Vendor specific macro information directives that do not fit into one of the standard categories.

Define and Undefine Entries

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

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

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

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

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

Start File Entries

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

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

End File Entries

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

Vendor Extension Entries

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

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

Base Source Entries

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

Macinfo Entries for Command Line Options

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

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

General Rules and Restrictions

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

All macinfo entries representing command line options should appear in the same order as the relevant command line options were given to the compiler. In the case where the compiler itself implicitly supplies one or more macro definitions or un-definitions in addition to those which may be specified on the command line, macinfo entries should also be pro-

duced for these implicit definitions and un-definitions, and these entries should also appear in the proper order relative to each other and to any definitions or undefinitions given explicitly by the user on the command line.

Call Frame Information

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

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

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

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

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

- Prologue and epilogue code is not always in distinct blocks at the beginning and end of a subroutine. It is common to duplicate the epilogue code at the site of each return from the code. Sometimes a compiler breaks up the register save/unsave operations and moves them into the body of the subroutine to just where they are needed.
- Compilers use different ways to manage the call frame. Sometimes they use a frame pointer register, sometimes not.
- The algorithm to compute the CFA changes as you progress through the prologue and epilogue code. (By definition, the CFA value does not change.)
- Some subroutines have no call frame.

- Sometimes a register is saved in another register that by convention does not need to be saved.
- Some architectures have special instructions that perform some or all of the register management in one instruction, leaving special information on the stack that indicates how registers are saved.
- Some architectures treat return address values specially. For example, in one architecture, the call instruction guarantees that the low order two bits will be zero and the return instruction ignores those bits. This leaves two bits of storage that are available to other uses that must be treated specially.

Structure of Call Frame Information

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

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

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

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

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

The register rules are:

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

architectural	The rule is defined externally to this specification by the augmenter.

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

The virtual unwind information is encoded in a self-contained section called.debug_frame. Entries in a.debug_frame section are aligned on an addressing unit boundary and come in two forms: A Common Information Entry (CIE) and a Frame Description Entry (FDE). Sizes of data objects used in the encoding of the .debug_frame section are described in terms of the same data definitions used for the line number information (see “Definitions” on page 24-49).

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

1. length
   A uword constant that gives the number of bytes of the CIE structure, not including the length field, itself (length mod<addressing unit size>== 0).

2. CIE_id
   A uword constant that is used to distinguish CIEs from FDEs.

3. version
   A ubyte version number. This number is specific to the call frame information and is independent of the DWARF version number.

4. augmentation
   A null terminated string that identifies the augmentation to this CIE or to the FDEs that use it. If a reader encounters an augmentation string that is unexpected, then only the following fields can be read: CIE:length,CIE_id,version,augmentation; FDE:length, CIE_pointer, initial_location, address_range. If there is no augmentation, this value is a zero byte.

5. code_alignment_factor
   An unsigned LEB128 constant that is factored out of all advance location instructions (see below).

6. data_alignment_factor
   A signed LEB128 constant that is factored out of all offset instructions (see below.)

7. return_address_register
   A ubyte constant that indicates which column in the rule table represents the return address of the function. Note that this column might not correspond to an actual machine register.

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

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

An FDE contains the following fields, in order:

1. length
   A uword constant that gives the number of bytes of the header and instruction stream for this function (not including the length field itself) (length mod<addressing unit size>==0).

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

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

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

5. instructions
   A sequence of table defining instructions that are described below.

Call Frame Instructions

Each call frame instruction is defined to take 0 or more operands. Some of the operands may be encoded as part of the opcode (see “Call Frame Information” on page 24-83). The instructions are as follows:

1. DW_CFA_advance_loctakes a single argument that represents a constant delta. The required action is to create a new table row with a location value that is computed by taking the current entry's location value and adding (delta *code_alignment_factor). All other values in the new row are initially identical to the current row.

2. DW_CFA_offset takes two arguments: an unsigned LEB128 constant representing a factored offset and a register number. The required action is to change the rule for the register indicated by the register number to be an offset(N) rule with a value of (N = factored offset *data_alignment_factor).

3. DW_CFA_restore takes a single argument that represents a register number. The required action is to change the rule for the indicated register to the rule assigned it by the initial_instructions in the CIE.

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

5. DW_CFA_advance_loc1takes a single ubyte argument that represents a constant delta. This instruction is identical to DW_CFA_advance_loc except for the encoding and size of the delta argument.

6. DW_CFA_advance_loc2takes a single uhalf argument that represents a constant delta. This instruction is identical to DW_CFA_advance_loc except for the encoding and size of the delta argument.

7. DW_CFA_advance_loc4takes a single uword argument that represents a constant delta. This instruction is identical to DW_CFA_advance_loc except for the encoding and size of the delta argument.

8. DW_CFA_offset_extended takes two unsigned LEB128 arguments representing a register number and a factored offset. This instruction is identical to DW_CFA_offset except for the encoding and size of the register argument.

9. DW_CFA_restore_extended takes a single unsigned LEB128 argument that represents a register number. This instruction is identical to DW_CFA_restore except for the encoding and size of the register argument.

10. DW_CFA_undefined takes a single unsigned LEB128 argument that represents a register number. The required action is to set the rule for the specified register to “undefined.”

11. DW_CFA_same_value takes a single unsigned LEB128 argument that represents a register number. The required action is to set the rule for the specified register to “same value.”

12. DW_CFA_register takes two unsigned LEB128 arguments representing register numbers. The required action is to set the rule for the first register to be the same as the rule for the second register.

13. DW_CFA_remember_state

14. DW_CFA_restore_state
    These instructions define a stack of information. Encountering the DW_CFA_remember_stateinstruction means to save the rules for every register on the current row on the stack. Encountering the DW_CFA_restore_state instruction means to pop the set of rules off the stack and place them in the current row. (This operation is useful for compilers that move epilogue code into the body of a function.)

15. DW_CFA_def_cfa takes two unsigned LEB128 arguments representing a register number and an offset. The required action is to define the current CFA rule to use the provided register and offset.

16. DW_CFA_def_cfa_register takes a single unsigned LEB128 argument representing a register number. The required action is to define the current CFA rule to use the provided register (but to keep the old offset).

17. DW_CFA_def_cfa_offset takes a single unsigned LEB128 argument representing an offset. The required action is to define the current CFA rule to use the provided offset (but to keep the old register).

18. DW_CFA_nop has no arguments and no required actions. It is used as padding to make the FDE an appropriate size.

Call Frame Instruction Usage

To determine the virtual unwind rule set for a given location (L1), one searches through the FDE headers looking at theinitial_locationandaddress_range values to see if L1 is contained in the FDE. If so, then:

1. Initialize a register set by reading theinitial_instructionsfield of the associated CIE.
2. Read and process the FDE's instruction sequence until a DW_CFA_advance_loc, DW_CFA_set_loc, or the end of the instruction stream is encountered.
3. If a DW_CFA_advance_loc or DW_CFA_set_loc instruction was encountered, then compute a new location value (L2). If L1>= L2 then process the instruction and go back to step 2.
4. The end of the instruction stream can be thought of as a DW_CFA_set_loc(initial_location + address_range) instruction. Unless the FDE is ill-formed, L1 should be less than L2 at this point.

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

For an example, see "Appendix 5 -- Call Frame Information Examples" on page 24-102.

Data Representation

This section describes the binary representation of the debugging information entry itself, of the attribute types and of other fundamental elements described above.

Vendor Extensibility

To reserve a portion of the DWARF name space and ranges of enumeration values for use for vendor specific extensions, special labels are reserved for tag names, attribute names, base type encodings, location operations, language names, calling conventions and call frame instructions.

The labels denoting the beginning and end of the reserved value range for vendor specific extensions consist of the appropriate prefix (DW_TAG DW_AT, DW_ATE, DW_OP, DW_LANG, or DW_CFA respectively) followed by _lo_user or _hi_user. For example, for entry tags, the special labels are DW_TAG_lo_user and DW_TAG_hi_user. Values in the range between prefix _lo_user and prefix _hi_user inclusive, are reserved for vendor specific extensions. Vendors may use values in this range without conflicting with current or future system-defined values. All other values are reserved for use by the system.

Vendor defined tags, attributes, base type encodings, location atoms, language names, calling conventions and call frame instructions, conventionally use the form prefix _vendor_id_name, where vendor_id is some identifying character sequence chosen so as to avoid conflicts with other vendors.

To ensure that extensions added by one vendor may be safely ignored by consumers that do not understand those extensions, the following rules should be followed:

1. New attributes should be added in such a way that a debugger may recognize the format of a new attribute value without knowing the content of that attribute value.
2. The semantics of any new attributes should not alter the semantics of previously existing attributes.
3. The semantics of any new tags should not conflict with the semantics of previously existing tags.

Reserved Error Values

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

Executable Objects and Shared Objects

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

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

File Constraints

All debugging information entries in a relocatable object file, executable object or shared object are required to be physically contiguous.

Format of Debugging Information

For each compilation unit compiled with a DWARF Version 2 producer, a contribution is made to the .debug_info section of the object file. Each such contribution consists of a compilation unit header followed by a series of debugging information entries. Unlike the information encoding for DWARF Version 1, Version 2 debugging information entries do not themselves contain the debugging information entry tag or the attribute name and form

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

This encoding was based on the observation that typical DWARF producers produce a very limited number of different types of debugging information entries. By extracting the common information from those entries into a separate table, we are able to compress the generated information.

Compilation Unit Header

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

1. A 4-byte unsigned integer representing the length of the .debug_info contribution for that compilation unit, not including the length field itself.
2. A 2-byte unsigned integer representing the version of the DWARF information for that compilation unit. For DWARF Version 2, the value in this field is 2.
3. A 4-byte unsigned offset into the .debug_abbrev section. This offset associates the compilation unit with a particular set of debugging information entry abbreviations.
4. A 1-byte unsigned integer representing the size in bytes of an address on the target architecture. If the system uses segmented addressing, this value represents the size of the offset portion of an address.

The compilation unit header does not replace the DW_TAG_compile_unit debugging information entry. It is additional information that is represented outside the standard DWARF tag/attributes format.

Debugging Information Entry

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

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

Abbreviation Tables

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

The abbreviation table for a single compilation unit consists of a series of abbreviation declarations. Each declaration specifies the tag and attributes for a particular form of debugging information entry. Each declaration begins with an unsigned LEB128 number representing the abbreviation code itself. It is this code that appears at the beginning of a debugging information entry in the .debug_info section. As described above, the abbreviation code 0 is reserved for null debugging information entries. The abbreviation code is followed by another unsigned LEB128 number that encodes the entry's tag. The encodings for the tag names are given in Table 24-14 and Table 24-15.

Following the tag encoding is a 1-byte value that determines whether a debugging information entry using this abbreviation has child entries or not. If the value is DW_CHILDREN_yes, the next physically succeeding entry of any debugging information entry using this abbreviation is the first child of the prior entry. If the 1-byte value following the abbreviation's tag encoding is DW_CHILDREN_no, the next physically succeeding entry of any debugging information entry using this abbreviation is a sibling of the prior entry. (Either the first child or sibling entries may be null entries). The encodings for the child determination byte are given in Table 24-16. (As mentioned in “Relationship of Debugging Information Entries” on page 24-7, each chain of sibling entries is terminated by a null entry).

Finally, the child encoding is followed by a series of attribute specifications. Each attribute specification consists of two parts. The first part is an unsigned LEB128 number representing the attribute's name. The second part is an unsigned LEB128 number representing the attribute's form. The series of attribute specifications ends with an entry containing 0 for the name and 0 for the form.

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

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

See “Appendix 2 -- Organization of Debugging Information” on page 24-96 for a depiction of the organization of the debugging information.

Attribute Encodings

The encodings for the attribute names are given in Table 24-17 and Table 24-18.

The attribute form governs how the value of the attribute is encoded. The possible forms may belong to one of the following form classes:

address — Represented as an object of appropriate size to hold an address on the target machine (DW_FORM_addr). This address is relocatable in a relocatable object file and is relocated in an executable file or shared object.

block Blocks come in four forms. The first consists of a 1-byte length followed by 0 to 255 contiguous information bytes (DW_FORM_block1). The second consists of a 2-byte length followed by 0 to 65,535 contiguous information bytes (DW_FORM_block2). The third consists of a 4-byte

Table 24-14. Tag Encodings (Part 1)

Tag name	Value
DW_TAG_array_type	0x01
DW_TAG_class_type	0x02
DW_TAG_entry_point	0x03
DW_TAG_enumeration_type	0x04
DW_TAG_formal_parameter	0x05
DW_TAG_imported_declaration	0x08
DW_TAG_label	0x0a
DW_TAG_lexical_block	0x0b
DW_TAG_member	0x0d
DW_TAG_pointer_type	0x0f
DW_TAG_reference_type	0x10
DW_TAG_compile_unit	0x11
DW_TAG_string_type	0x12
DW_TAG_structure_type	0x13
DW_TAG_subroutine_type	0x15
DW_TAG_typedef	0x16
DW_TAG_union_type	0x17
DW_TAG_unspecified_parameters	0x18
DW_TAG_variant	0x19
DW_TAG_common_block	0x1a
DW_TAG_common_inclusion	0x1b
DW_TAG_inheritance	0x1c
DW_TAG_inlined_subroutine	0x1d
DW_TAG_module	0x1e
DW_TAG_ptr_to_member_type	0x1f
DW_TAG_set_type	0x20
DW_TAG_subrange_type	0x21
DW_TAG_with_stmt	0x22
DW_TAG_access_declaration	0x23

Table 24-14. Tag Encodings (Part 1) (Cont.)

Tag name	Value
DW_TAG_base_type	0x24
DW_TAG_catch_block	0x25
DW_TAG_const_type	0x26
DW_TAG_constant	0x27
DW_TAG_enumerator	0x28
DW_TAG_file_type	0x29

length followed by 0 to 4,294,967,295 contiguous information bytes (DW_FORM_block4). The fourth consists of an unsigned LEB128 length followed by the number of bytes specified by the length (DW_FORM_block). In all forms, the length is the number of information bytes that follow. The information bytes may contain any mixture of relocated (or relocatable) addresses, references to other debugging information entries or data bytes.

constant — There are six forms of constants: one, two, four and eight byte values (respectively, DW_FORM_data1, DW_FORM_data2, DW_FORM_data4, and DW_FORM_data8). There are also variable

Table 24-15. Tag Encodings (Part 2)

Tag name	Value
DW_TAG_friend	0x2a
DW_TAG_namelist	0x2b
DW_TAG_namelist_item	0x2c
DW_TAG_packed_type	0x2d
DW_TAG_subprogram	0x2e
DW_TAG_template_type_param	0x2f
DW_TAG_template_value_param	0x30
DW_TAG_thrown_type	0x31
SDW_TAG_try_block	0x32
DW_TAG_variant_part	0x33
DW_TAG_variable	0x34
DW_TAG_volatile_type	0x35
DW_TAG_lo_user	0x4080
DW_TAG_hi_user	0xffff

Table 24-16. Child Determination Encodings

Child determination name	Value
DW_CHILDREN_no	0
DW_CHILDREN_yes	1

length constant data forms encoded using LEB128 numbers (see below). Both signed (DW_FORM_sdata) and unsigned (DW_FORM_udata) variable length constants are available.

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

reference — There are two types of reference. The first is an offset relative to the first byte of the compilation unit header for the compilation unit containing the reference. The offset must refer to an entry within that same compilation unit. There are five forms for this type of reference: one, two, four and eight byte offsets (respectively, DW_FORM_ref1, DW_FORM_ref2, DW_FORM_ref4, and DW_FORM_ref8). There are is also an unsigned variable length offset encoded using LEB128 numbers (DW_FORM_ref_udata).

The second type of reference is the address of any debugging information entry within the same executable or shared object; it may refer to an entry in a different compilation unit from the unit containing the reference. This type of reference (DW_FORM_ref_addr) is the size of an address on the target architecture; it is relocatable in a relocatable object file and relocated in an executable file or shared object.

The use of compilation unit relative references will reduce the number of link-time relocations and so speed up linking.

The use of address-type references allows for the commonization of information, such as types, across compilation units.

Table 24-17. Attribute Encodings (Part 1)

Attribute name	Value	Classes
DW_AT_sibling	0x01	reference
DW_AT_location	0x02	block, constant
DW_AT_name	0x03	string
DW_AT_ordering	0x09	constant
DW_AT_byte_size	0x0b	constant
DW_AT_bit_offset	0x0c	constant
DW_AT_bit_size	0x0d	constant
DW_AT_stmt_list	0x10	constant

Table 24-17. Attribute Encodings (Part 1) (Cont.)

Attribute name	Value	Classes
DW_AT_low_pc	0x11	address
DW_AT_high_pc	0x12	address
DW_AT_language	0x13	constant
DW_AT_discr	0x15	reference
DW_AT_discr_value	0x16	block
DW_AT_visibility	0x17	constant
DW_AT_import	0x18	reference
DW_AT_string_length	0x19	block, constant
DW_AT_common_reference	0x1a	reference
DW_AT_comp_dir	0x1b	string
DW_AT_const_value	0x1c	string, constant, block
DW_AT_containing_type	0x1d	reference
DW_AT_default_value	0x1e	reference
DW_AT_inline	0x20	constant
DW_AT_is_optional	0x21	flag
DW_AT_lower_bound	0x22	constant, reference
DW_AT_producer	0x25	string
DW_AT_prototyped	0x27	flag
DW_AT_return_addr	0x2a	block, constant
DW_AT_start_scope	0x2c	constant
DW_AT_stride_size	0x2e	constant
DW_AT_upper_bound	0x2f	constant, reference

string A string is a sequence of contiguous non-null bytes followed by one null byte. A string may be represented immediately in the debugging information entry itself (DW_FORM_string), or may be represented as a 4-byte offset into a string table contained in the .debug_str section of the object file (DW_FORM_strp).

The form encodings are listed in Table 24-19.

Variable Length Data

The special constant data forms DW_FORM_sdata and DW_FORM_udata are encoded using “Little Endian Base 128” (LEB128) numbers. LEB128 is a scheme for encoding integers densely that exploits the assumption that most integers are small in magnitude.

(This encoding is equally suitable whether the target machine architecture represents data in big-endian or little-endian order. It is "little endian" only in the sense that it avoids using space to represent the "big" end of an unsigned integer, when the big end is all zeroes or sign extension bits).

Table 24-18. Attribute Encodings (Part 2)

Attribute name	Value	Classes
DW_AT_abstract_origin	0x31	reference
DW_AT_accessibility	0x32	constant
DW_AT_address_class	0x33	constant
DW_AT_artificial	0x34	flag
DW_AT_base_types	0x35	reference
DW_AT_calling_convention	0x36	constant
DW_AT_count	0x37	constant, reference
DW_AT_data_member_location	0x38	block, reference
DW_AT_decl_column	0x39	constant
DW_AT_decl_file	0x3a	constant
DW_AT_decl_line	0x3b	constant
DW_AT_declaration	0x3c	flag
DW_AT_discr_list	0x3d	block
DW_AT_encoding	0x3e	constant
DW_AT_external	0x3f	flag
DW_AT_frame_base	0x40	block, constant
DW_AT_friend	0x41	reference
DW_AT_identifier_case	0x42	constant
DW_AT_macro_info	0x43	constant
DW_AT_namelist_item	0x44	block
DW_AT_priority	0x45	reference
DW_AT_segment	0x46	block, constant
DW_AT_specification	0x47	reference
DW_AT_static_link	0x48	block, constant
DW_AT_type	0x49	reference
DW_AT_use_location	0x4a	block, constant
DW_AT_variable_parameter	0x4b	flag
DW_AT_virtuality	0x4c	constant

Table 24-18. Attribute Encodings (Part 2) (Cont.)

Attribute name	Value	Classes
DW_AT_vtable_elem_location	0x4d	block, reference
DW_AT_lo_user	0x2000	--
DW_AT_hi_user	0x3fff	--

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

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

Table 24-20 gives some examples of DW_FORM_udata numbers. The0x80in each case is the high order bit of the byte, indicating that an additional byte follows:

The encoding for DW_FORM_sdata (signed, 2s complement LEB128) numbers is similar, except that the criterion for discarding high order bytes is not whether they are zero, but whether they consist entirely of sign extension bits. Consider the 32-bit integer-2. The three high level bytes of the number are sign extension, thus LEB128 would represent it as a single byte

Table 24-19. Attribute Form Encodings

Form name	Value	Class
DW_FORM_addr	0x01	address
DW_FORM_blocK2	0x03	block
DW_FORM_blocK4	0x04	block
DW_FORM_data2	0x05	constant
DW_FORM_data4	0x06	constant
DW_FORM_data8	0x07	constant
DW_FORM_string	0x08	string
DW_FORM_block	0x09	block
DW_FORM_block1	0x0a	block
DW_FORM_data1	0x0b	constant
DW_FORM_flag	0x0c	flag
DW_FORM_sdata	0x0d	constant
DW_FORM_strp	0x0e	string
DW_FORM_udata	0x0f	constant

Table 24-19. Attribute Form Encodings (Cont.)

Form name	Value	Class
DW_FORM_ref_addr	0x10	reference
DW_FORM_ref1	0x11	reference
DW_FORM_ref2	0x12	reference
DW_FORM_ref4	0x13	reference
DW_FORM_ref8	0x14	reference
DW_FORM_ref_udata	0x15	reference
DW_FORM_indirect	0x16	(see "Abbreviation Tables" on page 24-67)

Table 24-20. Examples of unsigned LEB128 Encodings

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

containing the low order 7 bits, with the high order bit cleared to indicate the end of the byte stream. Note that there is nothing within the LEB128 representation that indicates whether an encoded number is signed or unsigned. The decoder must know what type of number to expect.

Table 24-21 gives some examples of DW_FORM_sdata numbers.

"Appendix 4 -- Encoding and decoding variable length data" on page 24-100 gives algorithms for encoding and decoding these forms.

Location Descriptions

Location Expressions

A location expression is stored in a block of contiguous bytes. The bytes form a set of operations. Each location operation has a 1-byte code that identifies that operation. Operations can be followed by one or more bytes of additional data. All operations in a location. expression are concatenated from left to right. The encodings for the operations in a location expression are described in Table 24-22 and Table 24-23.

Table 24-21. Examples of signed LEB128 Encodings

Number	First byte	Second byte
2	2	--
-2	0x7e	--
127	127+0x80	0
-127	1+0x80	0x7f
128	0+0x80	1
-128	0+0x80	0x7f
129	1+0x80	1
-129	0x7f+0x80	0x7e

Table 24-22. Location Operation Encodings (Part 1)

Operation	Code	No. of Operands	Notes
DW_OP_addr	0x03 1	constant address (size target specific)	
DW_OP_deref	0x06	0	
DW_OP_const1u	0x08	1	1-byte constant
DW_OP_const1s	0x09	1	1-byte constant
DW_OP_const2u	0x0a	1	2-byte constant
DW_OP_const2s	0x0b	1	2-byte constant
DW_OP_const4u	0x0c	1	4-byte constant
DW_OP_const4s	0x0d	1	4-byte constant
DW_OP_const8u	0x0e	1	8-byte constant
DW_OP_const8s	0x0f	1	8-byte constant
DW_OP_constu	0x10	1	ULEB128 constant
DW_OP_consts	0x11	1	SLEB128 constant
DW_OP_dup	0X12	0	
DW_OP_drop	0X13	0	
DW_OP_over	0X14	0	
DW_OP_pick	0X15	1	1-byte stack index
DW_OP_swap	0X16	0	
DW_OP_rot	0X17	0	

Table 24-22. Location Operation Encodings (Part 1) (Cont.)

Operation	Code	No. of Operands	Notes
DW_OP_xderef	0x18	0	
DW_OP_abs	0X19	0	
DW_OP_and	0X1a	0	
DW_OP_div	0X1b	0	
DW_OP_minus	0x1c	0	
DW_OP_mod	0X1d	0	
DW_OP_mul	0X1e	0	
DW_OP_neg	0X1f	0	
DW_OP_not	0X20	0	
DW_OP_or	0X21	0	
DW_OP_plus	0X22	0	
DW_OP_plus_uconst	0x23	1	ULEB128 addend
DW_OP_shl	0X24	0	
DW_OP_shr	0X25	0	
DW_OP_shra	0X26	0	

Table 24-23. Location Operation Encodings (Part 2)

Operation	Code	No. of Operands	Notes
DW_OP_xor	0X27	0	
DW_OP_skip	0X2f	1	signed 2-byte constant
DW_OP_bra	0X28	1	signed 2-byte constant
DW_OP_eq	0X29	0	
DW_OP_ge	0X2A	0	
DW_OP_gt	0X2B	0	
DW_OP_le	0X2C	0	
DW_OP_lt	0X2D	0	
DW_OP_ne	0X2E	0	
DW_OP_lit0	0X30	0	literals 0..31 = (DW_OP_LIT0|literal)
DW_OP_lit1	0X31	0	
...			
DW_OP_lit31	0x4f	0	

Table 24-23. Location Operation Encodings (Part 2) (Cont.)

Operation	Code	No. of Operands	Notes
DW_OP_reg0	0X50	0	reg 0..31 = (DW_OP_REG0|reg-num)
DW_OP_reg1	0X51	0	
...			
DW_OP_reg31	0x6f	0	
DW_OP_breg0	0x70	1	SLEB128 offset
DW_OP_breg1	0x71	1	base reg 0..31 = (DW_OP_BREG0|regnum)
...			
DW_OP_breg31	0x8f	1	
DW_OP_regx	0X90	1	ULEB128 register
DW_OP_fbreg	0x91	1	SLEB128 offset
DW_OP_bregx	0x92	2	ULEB128 register followed by SLEB128 offset
DW_OP_piece	0x93	1	ULEB128 size of piece addressed
DW_OP_deref_size	0X94	1	1-byte size of data retrieved
DW_OP_xderef_size	0X95	1	1-byte size of data retrieved
DW_OP_nop	0X96	0	
DW_OP_lo_user	0xe0		
DW_OP_hi_user	0xff		

Location Lists

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

Base Type Encodings

The values of the constants used in the DW_AT_encoding attribute are given in Table 24-24.

Accessibility Codes

The encodings of the constants used in the DW_AT_accessibility attribute are given in Table 24-25.

Table 24-24. Base Type Encoding Values

Base type encoding name	Value
DW_ATE_address	0x1
DW_ATE_boolean	0x2
DW_ATE_complex_float	0x3
DW_ATE_float	0x4
DW_ATE_signed	0x5
DW_ATE_signed_char	0x6
DW_ATE_unsigned	0x7
DW_ATE_unsigned_char	0x8
DW_ATE_lo_user	0x80
DW_ATE_hi_user	0xff

Table 24-25. Accessibility Encodings

Accessibility code name	Value
DW_ACCESS_public	1
DW_ACCESS_protected	2
DW_ACCESS_private	3

Visibility Codes

The encodings of the constants used in the DW_AT_visibility attribute are given in Table 24-26.

Table 24-26. Visibility Encodings

Visibility code name	Value
DW_VIS_local	1
DW_VIS_exported	2
DW_VIS_qualified	3

Virtuality Codes

The encodings of the constants used in the DW_AT_virtuality attribute are given in Table 24-27.

Table 24-27. Virtuality Encodings

Virtuality code name	Value
DW_VIRTUALITY_none	0
DW_VIRTUALITY_virtual	1
DW_VIRTUALITY_pure_virtual	2

Source Languages

The encodings for source languages are given in Table 24-28. Names marked with ??? and their associated values are reserved, but the languages they represent are not supported in DWARF Version 2.

Address Class Encodings

The value of the common address class encoding DW_ADDR_none is 0.

Table 24-28. Language Encodings

Language name	Value
DW_LANG_C89	0x0001
DW_LANG_C	0x0002
DW_LANG_Ada83???	0x0003
DW_LANG_C_plus_plus	0x0004
DW_LANG_Cobol74???	0x0005
DW_LANG_Cobol85???	0x0006
DW_LANG_Fortran77	0x0007
DW_LANG_Fortran90	0x0008
SDW_LANG_Pascal83	0x0009S
DW_LANG_Modula2	0x000a
DW_LANG_lo_user	0x8000
DW_LANG_hi_user	0xffff

Identifier Case

The encodings of the constants used in the DW_AT_identifier_case attribute are given in Table 24-29.

Table 24-29. Identifier Case Encodings

Identifier Case Name	Value
DW_ID_case_sensitive	0
DW_ID_up_case	1
DW_ID_down_case	2
DW_ID_case_insensitive	3

Calling Convention Encodings

The encodings for the values of the DW_AT_calling_convention attribute are given in Table 24-30.

Table 24-30. Calling Convention Encodings

Calling Convention Name	Value
DW_CC_normal	0x1
DW_CC_program	0x2
DW_CC_nocall	0x3
DW_CC_lo_user	0x40
DW_CC_hi_user	0xff

Inline Codes

The encodings of the constants used in the DW_AT_inline attribute are given in Table 24-31.

Table 24-31. Inline Encodings

Inline Code Name	Value
DW_INL_not_inlined	0

Table 24-31. Inline Encodings (Cont.)

Inline Code Name	Value
DW_INL_inlined	1
DW_INL_declared_not_inlined	2
DW_INL_declared_inlined	3

Array Ordering

The encodings for the values of the order attributes of arrays is given in Table 24-32.

Table 24-32. Ordering Encodings

Ordering name	Value
DW_ORD_row_major	0
DW_ORD_col_major	1

Discriminant Lists

The descriptors used in the DW_AT_dicsr_list attribute are encoded as 1-byte constants.

The defined values are presented in Table 24-33.

Table 24-33. Discriminant Descriptor Encodings

Descriptor Name	Value
DW_DSC_label	0
DW_DSC_range	1

Name Lookup Table

Each set of entries in the table of global names contained in the .debug_pubnames section begins with a header consisting of: a 4-byte length containing the length of the set of entries for this compilation unit, not including the length field itself; a 1-byte version identifier containing the value 2 for DWARF Version 2; a 4-byte offset into the .debug_info section; and a 4-byte length containing the size in bytes of the contents of the .debug_info section generated to represent this compilation unit. This header is followed by a series of tuples. Each tuple consists of a 4-byte offset followed by a string of non-null bytes terminated by one null byte. Each set is terminated by a 4-byte word containing the value 0.

Address Range Table

Each set of entries in the table of address ranges contained in the .debug_aranges section begins with a header consisting of: a 4-byte length containing the length of the set of entries for this compilation unit, not including the length field itself; a 2-byte version identifier containing the value 2 for DWARF Version 2; a 4-byte offset into the .debug_info section; a 1-byte unsigned integer containing the size in bytes of an address (or the offset portion of an address for segmented addressing) on the target system; and a 1-byte unsigned integer containing the size in bytes of a segment descriptor on the target system. This header is followed by a series of tuples. Each tuple consists of an address and a length, each in the size appropriate for an address on the target architecture. The first tuple following the header in each set begins at an address that is a multiple of the size of a single tuple (that is, twice the size of an address). The header is padded, if necessary, to the appropriate boundary. Each set of tuples is terminated by a 0 for the address and 0 for the length.

Line Number Information

The sizes of the integers used in the line number and call frame information sections are as follows:

sbyte Signed 1-byte value.

ubyte Unsigned 1-byte value.

uhalf Unsigned 2-byte value.

sword Signed 4-byte value.

uword Unsigned 4-byte value.

The version number in the statement program prologue is 2 for DWARF Version 2. The boolean values “true” and “false” used by the statement information program are encoded as a single byte containing the value 0 for “false,” and a non-zero value for “true.” The encodings for the pre-defined standard opcodes are given in Table 24-34.

Table 24-34. Standard Opcode Encodings

Opcode Name	Value
DW_LNS_copy	1
DW_LNS_advance_pc	2
DW_LNS_advance_line	3
DW_LNS_set_file	4
DW_LNS_set_column	5
DW_LNS_negate_stmt	6

Table 24-34. Standard Opcode Encodings (Cont.)

Opcode Name	Value
DW_LNS_set_basic_block	7
DW_LNS_const_add_pc	8
DW_LNS_fixed_advance_pc	9

The encodings for the pre-defined extended opcodes are given in Table 24-35.

Table 24-35. Extended Opcode Encodings

Opcode Name	Value
DW_LNE_end_sequence	1
DW_LNE_set_address	2
DW_LNE_define_file	3

Macro Information

The source line numbers and source file indices encoded in the macro information section are represented as unsigned LEB128 numbers as are the constants in an DW_MACINFO_vend_ext entry. The macinfo type is encoded as a single byte. The encodings are given in Table 24-36.

Table 24-36. Macinfo Type Encodings

Macinfo Type Name	Value
DW_MACINFO_define	1
DW_MACINFO_undef	2
DW_MACINFO_start_file	3
DW_MACINFO_end_file	4
DW_MACINFO_vend_ext	255

Call Frame Information

The value of the CIE id in the CIE header is0xffffffff. The initial value of the CIE version number is 1.

Call frame instructions are encoded in one or more bytes. The primary opcode is encoded in the high order two bits of the first byte (that is, opcode = byte>>6). An operand or

extended opcode may be encoded in the low order 6 bits. Additional operands are encoded in subsequent bytes.

The instructions and their encodings are presented in Table 24-37.

Table 24-37. Call Frame Instruction Encodings

Instruction	High 2 Bits	Low 6 Bits	Operand 1	Operand 2
DW_CFA_advance_loc	0x1	delta		
DW_CFA_offset	0x2	register	ULEB128offset	
DW_CFA_restore	0x3	register		
DW_CFA_set_loc	0	0x01	address	
DW_CFA_advance_loc1	0	0x02	1-byte delta	
DW_CFA_advance_loc2	0	0x03	2-byte delta	
DW_CFA_advance_loc4	0	0x04	4-byte delta	
DW_CFA_offset_extended	0	0x05	ULEB128 register	ULEB128 offset
DW_CFA_restore_extended	0	0x06	ULEB128 register	
DW_CFA_undefined	0	0x07	ULEB128 register	
DW_CFA_same_value	0	0x08	ULEB128 register	
DW_CFA_register	0	0x09	ULEB128 register	ULEB128 register
DW_CFA_remember_state	0	0x0a		
DW_CFA_restore_state	0	0x0b		
DW_CFA_def_cfa	0	0x0c	ULEB128 register	ULEB128 offset
DW_CFA_def_cfa_register	0	0x0d	ULEB128 register	
DW_CFA_def_cfa_offset	0	0x0e	ULEB128 offset	
DW_CFA_nop	0	0		
DW_CFA_lo_user	0	0x1c		
DW_CFA_hi_user	0	0x3f		

Dependencies

The debugging information in this format is intended to exist in the .debug_abbrev, .debug_aranges, .debug_frame, .debug_info, .debug_line, .debug_loc, .debug_macinfo, .debug_pubnames and .debug_str sections of an object file. The information is not word-aligned, so the assembler must provide a way for the compiler to produce 2-byte and 4-byte quantities without alignment restrictions, and the linker must be able to relocate a 4-byte reference at an arbitrary alignment. In target architectures with 64-bit addresses, the assembler and linker must similarly handle 8-byte references at arbitrary alignments.

Future Directions

The UNIX International Programming Languages SIG is working on a specification for a set of interfaces for reading DWARF information, that will hide changes in the representation of that information from its consumers. It is hoped that using these interfaces will make the transition from DWARF Version 1 to Version 2 much simpler and will make it easier for a single consumer to support objects using either Version 1 or Version 2 DWARF.

A draft of this specification is available for review from UNIX International. The Programming Languages SIG wishes to stress, however, that the specification is still in flux.

Appendix 1 -- Current Attributes by Tag Value

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

Table 24-38. Current Attributes by Tag Value

TAG NAME	APPLICABLE ATTRIBUTES
DW_TAG_access_declaration	DECL???
	DW_AT_accessibility
	DW_AT_name
	DW_AT_sibling
DW_TAG_array_type	DECL
	DW_AT_abstract_origin
	DW_AT_accessibility
	DW_AT_byte_size
	DW_AT_declaration
	DW_AT_name
	DW_AT_ordering
	DW_AT_sibling
	DW_AT_start_scope
	DW_AT_stride_size
	DW_AT_type

Table 24-38. Current Attributes by Tag Value (Cont.)

TAG NAME	APPLICABLE ATTRIBUTES
	DW_AT_visibility
DW_TAG_base_type	DW_AT_bit_offset
	DW_AT_bit_size
	DW_AT_byte_size
	DW_AT_encoding
	DW_AT_name
	DW_AT_sibling
DW_TAG_catch_block	DW_AT_abstract_origin
	DW_AT_high_pc
	DW_AT_low_pc
	DW_AT_segment
	DW_AT_sibling
DW_TAG_class_type	DECL
	DW_AT_abstract_origin
	DW_AT_accessibility
	DW_AT_byte_size
	DW_AT_declaration
	DW_AT_name
	DW_AT_sibling
	DW_AT_start_scope
	DW_AT_visibility
DW_TAG_common_block	DECL
	DW_AT_declaration
	DW_AT_location
	DW_AT_name
	DW_AT_sibling
	DW_AT_visibility
DW_TAG_common_inclusion	DECL
	DW_AT_common_reference
	DW_AT_declaration
	DW_AT_sibling
	DW_AT_visibility
DW_TAG_compile_unit	DW_AT_base_types

Table 24-38. Current Attributes by Tag Value (Cont.)

TAG NAME	APPLICABLE ATTRIBUTES
	DW_AT_comp_dir
	DW_AT_identifier_case
	DW_AT_high_pc
	DW_AT_language
	DW_AT_low_pc
	DW_AT_macro_info
	DW_AT_name
	DW_AT_producer
	DW_AT_sibling
	DW_AT_stmt_list
DW_TAG_const_type	DW_AT_sibling
	DW_AT_type
DW_TAG_constant	DECL
	DW_AT_accessibility
	DW_AT_constant_value
	DW_AT_declaration
	DW_AT_external
	DW_AT_name
	DW_AT_sibling
	DW_AT_start_scope
	DW_AT_type
	DW_AT_visibility
DW_TAG_entry_point	DW_AT_address_class
	DW_AT_low_pc
	DW_AT_name
	DW_AT_return_addr
	DW_AT_segment
	DW_AT_sibling
	DW_AT_static_link
	DW_AT_type
DW_TAG_enumeration_type	DECL
	DW_AT_abstract_origin
	DW_AT_accessibility

Table 24-38. Current Attributes by Tag Value (Cont.)

TAG NAME	APPLICABLE ATTRIBUTES
	DW_AT_byte_size
	DW_AT_declaration
	DW_AT_name
	DW_AT_sibling
	DW_AT_start_scope
	DW_AT_visibility
DW_TAG_enumerator	DECLS
	DW_AT_const_value
	DW_AT_name
	DW_AT_sibling
DW_TAG_file_type	DECL
	DW_AT_abstract_origin
	DW_AT_byte_size
	DW_AT_name
	DW_AT_sibling
	DW_AT_start_scope
	DW_AT_type
	DW_AT_visibility
DW_TAG_formal_parameter	DECL
	DW_AT_abstract_origin
	DW_AT_artificial
	DW_AT_default_value
	DW_AT_is_optional
	DW_AT_location
	DW_AT_name
	DW_AT_segment
	DW_AT_sibling
	DW_AT_type
	DW_AT_variable_parameter
DW_TAG_friend	DECL
	DW_AT_abstract_origin
	DW_AT_friend
	DW_AT_sibling

Table 24-38. Current Attributes by Tag Value (Cont.)

TAG NAME	APPLICABLE ATTRIBUTES
DW_TAG_imported_declaration	DECL
	DW_AT_accessibility
	DW_AT_import
	DW_AT_name
	DW_AT_sibling
	DW_AT_start_scope
DW_TAG_inheritance	DECL
	DW_AT_accessibility
	DW_AT_data_member_location
	DW_AT_sibling
	DW_AT_type
	DW_AT_virtuality
DW_TAG_inlined_subroutine	DECL
	DW_AT_abstract_origin
	DW_AT_high_pc
	DW_AT_low_pc
	DW_AT_segment
	DW_AT_sibling
	DW_AT_return_addr
	DW_AT_start_scope
DW_TAG_label	DW_AT_abstract_origin
	DW_AT_low_pc
	DW_AT_name
	DW_AT_segment
	DW_AT_start_scope
	DW_AT_sibling
DW_TAG_lexical_block	DW_AT_abstract_origin
	DW_AT_high_pc
	DW_AT_low_pc
	DW_AT_name
	DW_AT_segment
	DW_AT_sibling
DW_TAG_member	DECL

Table 24-38. Current Attributes by Tag Value (Cont.)

TAG NAME	APPLICABLE ATTRIBUTES
	DW_AT_accessibility
	DW_AT_byte_size
	DW_AT_bit_offset
	DW_AT_bit_size
	DW_AT_data_member_location
	DW_AT_declaration
	DW_AT_name
	DW_AT_sibling
	DW_AT_type
	DW_AT_visibility
DW_TAG_module	DECL
	DW_AT_accessibility
	DW_AT_declaration
	DW_AT_high_pc
	DW_AT_low_pc
	DW_AT_name
	DW_AT_priority
	DW_AT_segment
	DW_AT_sibling
	DW_AT_visibility
DW_TAG_namelist	DECL
	DW_AT_accessibility
	DW_AT_abstract_origin
	DW_AT_declaration
	DW_AT_sibling
	DW_AT_visibility
DW_TAG_namelist_item	DECL
	DW_AT_namelist_item
	DW_AT_sibling
DW_TAG_packed_type	DW_AT_sibling
	DW_AT_type
DW_TAG_pointer_type	DW_AT_address_class
	DW_AT_sibling

Table 24-38. Current Attributes by Tag Value (Cont.)

TAG NAME	APPLICABLE ATTRIBUTES
	DW_AT_type
DW_TAG_ptr_to_member_type	DECL
	DW_AT_abstract_origin
	DW_AT_address_class
	DW_AT_containing_type
	DW_AT_declaration
	DW_AT_name
	DW_AT_sibling
	DW_AT_type
	DW_AT_use_location
	DW_AT_visibility
DW_TAG_reference_type	DW_AT_address_class
	DW_AT_sibling
	DW_AT_type
DW_TAG_set_type	DECL
	DW_AT_abstract_origin
	DW_AT_accessibility
	DW_AT_byte_size
	DW_AT_declaration
	DW_AT_name
	DW_AT_start_scope
	DW_AT_sibling
	DW_AT_type
	DW_AT_visibility
DW_TAG_string_type	DECL
	DW_AT_accessibility
	DW_AT_abstract_origin
	DW_AT_byte_size
	DW_AT_declaration
	DW_AT_name
	DW_AT_segment
	DW_AT_sibling
	DW_AT_start_scope

Table 24-38. Current Attributes by Tag Value (Cont.)

TAG NAME	APPLICABLE ATTRIBUTES
	DW_AT_string_length
	DW_AT_visibility
DW_TAG_structure_type	DECL
	DW_AT_abstract_origin
	DW_AT_accessibility
	DW_AT_byte_size
	DW_AT_declaration
	DW_AT_name
	DW_AT_sibling
	DW_AT_start_scope
	DW_AT_visibility
DW_TAG_subprogram	DECL
	DW_AT_abstract_origin
	DW_AT_accessibility
	DW_AT_address_class
	DW_AT_artificial
	DW_AT_calling_convention
	DW_AT_declaration
	DW_AT_external
	DW_AT_frame_base
	DW_AT_high_pc
	DW_AT_inline
	DW_AT_low_pc
	DW_AT_name
	DW_AT_prototyped
	DW_AT_return_addr
	DW_AT_segment
	DW_AT_sibling
	DW_AT_specification
	DW_AT_start_scope
	DW_AT_static_link
	DW_AT_type
	DW_AT_visibility

Table 24-38. Current Attributes by Tag Value (Cont.)

TAG NAME	APPLICABLE ATTRIBUTES
	DW_AT_virtuality
	DW_AT_vtable_elem_location
DW_TAG_subrange_type	DECL
	DW_AT_abstract_origin
	DW_AT_accessibility
	DW_AT_byte_size
	DW_AT_count
	DW_AT_declaration
	DW_AT_lower_bound
	DW_AT_name
	DW_AT_sibling
	DW_AT_type
	DW_AT_upper_bound
	DW_AT_visibility
DW_TAG_subroutine_type	DECL
	DW_AT_abstract_origin
	DW_AT_accessibility
	DW_AT_address_class
	DW_AT_declaration
	DW_AT_name
	DW_AT_prototyped
	DW_AT_sibling
	DW_AT_start_scope
	DW_AT_type
	DW_AT_visibility
DW_TAG_template_type_param	DECL
	DW_AT_name
	DW_AT_sibling
	DW_AT_type
DW_TAG_template_value_param	DECL
	DW_AT_name
	DW_AT_const_value
	DW_AT_sibling

Table 24-38. Current Attributes by Tag Value (Cont.)

TAG NAME	APPLICABLE ATTRIBUTES
	DW_AT_type
DW_TAG_thrown_type	DECL
	DW_AT_sibling
	DW_AT_type
DW_TAG_try_block	DW_AT_abstract_origin
	DW_AT_high_pc
	DW_AT_low_pc
	DW_AT_segment
	DW_AT_sibling
DW_TAG_typedef	DECL
	DW_AT_abstract_origin
	DW_AT_accessibility
	DW_AT_declaration
	DW_AT_name
	DW_AT_sibling
	DW_AT_start_scope
	DW_AT_type
	DW_AT_visibility
DW_TAG_union_type	DECL
	DW_AT_abstract_origin
	DW_AT_accessibility
	DW_AT_byte_size
	DW_AT_declaration
	DW_AT_friends
	DW_AT_name
	DW_AT_sibling
	DW_AT_start_scope
	DW_AT_visibility
DW_TAG_unspecified_parameters	DECL
	DW_AT_abstract_origin
	DW_AT_artificial
	DW_AT_sibling
DW_TAG_variable	DECL

Table 24-38. Current Attributes by Tag Value (Cont.)

TAG NAME	APPLICABLE ATTRIBUTES
	DW_AT_accessibility
	DW_AT_constant_value
	DW_AT_declaration
	DW_AT_external
	DW_AT_location
	DW_AT_name
	DW_AT_segment
	DW_AT_sibling
	DW_AT_specification
	DW_AT_start_scope
	DW_AT_type
	DW_AT_visibility
DW_TAG_variant	DECL
	DW_AT_accessibility
	DW_AT_abstract_origin
	DW_AT_declaration
	DW_AT_discr_list
	DW_AT_discr_value
	DW_AT_sibling
DW_TAG_variant_part	DECL
	DW_AT_accessibility
	DW_AT_abstract_origin
	DW_AT_declaration
	DW_AT_discr
	DW_AT_sibling
	DW_AT_type
DW_TAG_volatile_type	DW_AT_sibling
	DW_AT_type
DW_TAG_with_statement	DW_AT_accessibility
	DW_AT_address_class
	DW_AT_declaration
	DW_AT_high_pc
	DW_AT_location

Table 24-38. Current Attributes by Tag Value (Cont.)

TAG NAME	APPLICABLE ATTRIBUTES
	DW_AT_low_pc
	DW_AT_segment
	DW_AT_sibling
	DW_AT_type
	DW_AT_visibility

??? - DW_AT_decl_column, DW_AT_decl_file, DW_AT_decl_line.

Appendix 2 -- Organization of Debugging Information

The following diagram depicts the relationship of the abbreviation tables contained in the .debug_abbrev section to the information contained in the .debug_info section. Values are given in symbolic form, where possible.

Compilation Unit 1 - .debug_info

	length
	2
	a1 (abbreviation table offset)
	4
	1
	"myfile.c"
	"Best Compiler Corp: Version 1.3"
	"mymachine:/home/mydir/src:"
	DW_LANG_C89
	0x0
	0x55
	DW_FORM_data4
	0x0
e1:	2
	"char"
	DW_ATE_unsigned_char
	1
e2:	3
	e1
	4
	"POINTER"
	e2
	0

Compilation Unit 2 - .debug_info

length
2
a1 (abbreviation table offset)
4
...
4
"strp"
e2
...

Abbreviation Table - .debug_abbrev

a1:	1
	DW_TAG_compile_unit
	DW_CHILDREN_yes
	DW_AT_name DW_FORM_string
	DW_AT_producer DW_FORM_string
	DW_AT_ compdir DW_FORM_string
	DW_AT_language DW_FORM_data1
	DW_AT_low_poc DW_FORM_addr
	DW_AT_high_pc DW_FORM_addr
	DW_AT_stmt_list DW_FORM_indirect
	0 0
	2
	DW_TAG_base_type
	DW_CHILDREN_no
	DW_AT_name DW_FORM_string
	DW_AT_encoding DW_FORM_data1
	DW_AT_byte_size DW_FORM_data1
	0 0
	3
	DW_TAG_pointer_type
	DW_CHILDREN_no
	DW_AT_type DW_FORM_ref4

0	0
4	
DW_TAG_typedef	
DW_CHILDREN_no	
DW_AT_name	DW_FORM_string
DW_AT_type	DW_FORM_ref4
0	0
0	

Appendix 3 -- Statement Program Examples

Consider this simple source file and the resulting machine code for the Intel 8086 processor:

```
1:  int
2:  main()
    0x239: push pb
    0x23a: mov bp,sp
3:  {
4:  printf("Omit needless words\n");
    0x23c: mov ax,0xaa
    0x23f: push ax
    0x240: call _printf
    0x243: pop cx
5:  exit(0);
    0x244: xor ax,ax
    0x246: push ax
    0x247: call _exit
    0x24a: pop cx
6:  }
    0x24b: pop bp
    0x24c: ret
7:
    0x24d:
```

If the statement program prologue specifies the following:

```
minimum_instruction_length   1
opcode_base                  10
line_base                    1
line_range                   15
```

Then one encoding of the statement program would occupy 12 bytes (the opcode SPECIAL(m, n) indicates the special opcode generated for a line increment of m and an address increment of n):

Opcode	Operand	Byte Stream
DW_LNS_advance_pc	LEB128(0x239)	0x2, 0xb9, 0x04
SPECIAL(2, 0)		0xb
SPECIAL(2, 3)		0x38
SPECIAL(1, 8)		0x82
SPECIAL(1, 7)		0x73
DW_LNS_advance_pc	LEB128(2)	0x2, 0x2
DW_LNE_end_sequence		0x0, 0x1, 0x1

An alternate encoding of the same program using standard opcodes to advance the program counter would occupy 22 bytes:

Opcode	Operand	Byte Stream
DW_LNS_fixed_advance_pc	0x239	0x9, 0x39, 0x2
SPECIAL(2, 0)		0xb
DW_LNS_fixed_advance_pc	0x3	0x9, 0x3, 0x0
SPECIAL(2, 0)		0xb
DW_LNS_fixed_advance_pc	0x8	0x9, 0x8, 0x0
SPECIAL(1, 0)		0xa
DW_LNS_fixed_advance_pc	0x7	0x9, 0x7, 0x0
SPECIAL(1, 0)		0xa
DW_LNS_fixed_advance_pc	0x2	0x9, 0x2, 0x0
DW_LNE_end_sequence		0x0, 0x1, 0x1

Appendix 4 -- Encoding and decoding variable length data

Here are algorithms expressed in a C-like pseudo-code to encode and decode signed and unsigned numbers in LEB128:

Encode an unsigned integer:

```
do
{
    byte = low order 7 bits of value;
    value >>= 7;
    if (value != 0) /* more bytes to come */
        set high order bit of byte;
    emit byte;
} while (value != 0);
```

Encode a signed integer:

```
more = 1;
negative = (value < 0);
size = no. of bits in signed integer;
while(more)
{
    byte = low order 7 bits of value;
    value >>= 7;
    /* the following is unnecessary if the
     * implementation of >>= uses an arithmetic
     * rather than logical shift for a signed
     * left operand
     */
if (negative)
    /* sign extend */
    value |= - (1 << (size - 7));
/* sign bit of byte is 2nd high order bit (0x40) */
if ((value == 0 && sign bit of byte is clear) ||
    (value == -1 && sign bit of byte is set))
    more = 0;
else
    set high order bit of byte;
emit byte;
}
```

Decode unsigned LEB128 number:

```
result = 0;
shift = 0;
while(true)
{
    byte = next byte in input;
    result |= (low order 7 bits of byte << shift);
    if (high order bit of byte == 0)
        break;
    shift += 7;
}
```

Decode signed LEB128 number:

```
result = 0;
shift = 0;
size = no. of bits in signed integer;
while(true)
{
    byte = next byte in input;
    result |= (low order 7 bits of byte << shift);
    shift += 7;
    /* sign bit of byte is 2nd high order bit (0x40) */
    if (high order bit of byte == 0)
        break;
}
if ((shift < size) && (sign bit of byte is set))
    /* sign extend */
    result |= - (1 << shift);
```

Appendix 5 -- Call Frame Information Examples

The following example uses a hypothetical RISC machine in the style of the Motorola 88000.

- Memory is byte addressed.
- Instructions are all 4-bytes each and word aligned.
- Instruction operands are typically of the form:

  ```
  <destination reg> <source reg> <constant>
  ```

- The address for the load and store instructions is computed by adding the contents of the source register with the constant.
- There are 8 4-byte registers:

  R0 always 0
  R1 holds return address on call
  R2-R3 temp registers (not preserved on call)
  R4-R6 preserved on call
  R7 stack pointer.

- The stack grows in the negative direction.

The following are two code fragments from a subroutine called foo that uses a frame pointer (in addition to the stack pointer.) The first column values are byte addresses.

```
          ;; start prologue
foo       sub R7, R7, <fsize>          ; Allocate frame
foo+4     store R1, R7, (<fsize>-4)    ; Save the return address
foo+8     store R6, R7, (<fsize>-8)    ; Save R6
foo+12    add R6, R7, 0                ; R6 is now the Frame ptr
foo+16    store R4, R6, (<fsize>-12)   ; Save a preserve reg.
          ;; This subroutine does not change R5
          ...
          ;; Start epilogue (R7 has been returned to entry value)
foo+64    load R4, R6, (<fsize>-12)    ; Restore R4
foo+68    load R6, R7, (<fsize>-8)     ; Restore R6
foo+72    load R1, R7, (<fsize>-4)     ; Restore return address
foo+76    add R7, R7, <fsize>          ; Deallocate frame
foo+80    jump R                       ; Return
foo+84
```

The table for the foo subroutine is as follows. It is followed by the corresponding fragments from the .debug_frame section.

Loc	CFA	R0	R1	R2	R3	R4	R5	R6	R7	R8
foo	[R7]+0	s	u	u	u	s	s	s	s	
foo+4	[R7]+fsize	s	u	u	u	s	s	s	s	r1
foo+8	[R7]+fsize	s	u	u	u	s	s	s	s	c4
foo+12	[R7]+fsize	s	u	u	u	s	s	c8	s	c4
foo+16	[R6]+fsize	s	u	u	u	s	s	c8	s	c4
foo+20	[R6]+fsize	s	u	u	u	c12	s	c8	s	c4
foo+64	[R6]+fsize	s	u	u	u	c12	s	c8	s	c4
foo+68	[R6]+fsize	s	u	u	u	s	s	c8	s	c4
foo+72	[R7]+fsize	s	u	u	u	s	s	s	s	c4
foo+76	[R7]+fsize	s	u	u	u	s	s	s	s	r1
foo+80	[R7]+0	s	u	u	u	s	s	s	s	s

NOTES

1. R8 is the return address
2. s = same_value rule
3. u = undefined rule
4. rN = register(N) rule
5. cN = offset(N) rule

Common Information Entry (CIE):

cie	32	; length
cie+4	0xffffffff	; CIE_id
cie+8	1	; version
cie+9	0	; augmentation
cie+10	4	; code_alignment_factor
cie+11	4	; data_alignment_factor
cie+12	8	; R8 is the return addr.
cie+13	DW_CFA_def_cfa (7, 0)	; CFA = [R7]+0
cie+16	DW_CFA_same_value (0)	; R0 not modified (=0)
cie+18	DW_CFA_undefined (1)	; R1 scratch
cie+20	DW_CFA_undefined (2)	; R2 scratch
cie+22	DW_CFA_undefined (3)	; R3 scratch
cie+24	DW_CFA_same_value (4)	; R4 preserve
cie+26	DW_CFA_same_value (5)	; R5 preserve
cie+28	DW_CFA_same_value (6)	; R6 preserve
cie+30	DW_CFA_same_value (7)	; R7 preserve
cie+32	DW_CFA_register (8, 1)	; R8 is in R1
cie+35	DW_CFA_nop	; padding
cie+36	DW_CFA_nop	; padding
cie+37		

Frame Description Entry (FDE):

fde	44	; length
fde+4	cie	; CIE_ptr
fde+8	foo	; initial_location
fde+12	84	; address_range
fde+16	DW_CFA_advance_loc(1)	; instructions
fde+17	DW_CFA_def_cfa_offset(<fsize>/4)	; assuming <fsize> < 512
fde+19	DW_CFA_advance_loc(1)	
fde+20	DW_CFA_offset(8,1)	
fde+23	DW_CFA_advance_loc(1)	
fde+24	DW_CFA_offset(6,2)	
fde+27	DW_CFA_advance_loc(1)	
fde+28	DW_CFA_def_cfa_register(6)	
fde+30	DW_CFA_advance_loc(1)	
fde+31	DW_CFA_offset(4,3)	
fde+34	DW_CFA_advance_loc(12)	
fde+35	DW_CFA_restore(4)	
fde+36	DW_CFA_advance_loc(1)	
fde+37	DW_CFA_restore(6)	
fde+38	DW_CFA_def_cfa_register(7)	
fde+40	DW_CFA_advance_loc(1)	
fde+41	DW_CFA_restore(8)	
fde+42	DW_CFA_advance_loc(1)	
fde+43	DW_CFA_def_cfa_offset(0)	
fde+45	DW_CFA_nop	; padding
fde+46	DW_CFA_nop	; padding
fde+47	DW_CFA_nop	; padding
fde+48		

25
DWARF Access Library (libdwarf)

25
DWARF Access Library (libdwarf)

The material in this document represents work in progress of the UNIX International Programming Languages SIG.

Introduction

This document describes the libdwarf interface, a library of functions to provide access to DWARF debugging information records and DWARF line number information.

Purpose and Scope

As the DWARF information format evolves, the need exists for a functional interface to insulate client programs from the representation changes while preserving the relationship and semantics of current DWARF debugging information. The purpose of this document is to specify such an interface that shields DWARF consumers from the changes to the on-disk layout of DWARF debugging information. There is no effort made in this document to address the creation of DWARF debugging information records as that issue will be addressed in subsequent specifications.

Additionally, the focus of this document is the functional interface, and as such, implementation as well as optimization issues are intentionally ignored.

Definitions

DWARF debugging information entries (DIE) are the segments of information placed in the .debug* section by compilers, assemblers, and linkage editors that, in conjunction with line number entries, are necessary for symbolic source-level debugging. Refer to the document "DWARF Debugging Information Format" from UIPLSIG for a more complete description of these entries.

Line number entries are the information that is used to map executable statements to their corresponding location in the source file of their origin. Further information concerning line number entries can be found in the document cited above.

Overview

The remaining sections of this document describe the interface to libdwarf, first by describing the additional types defined by the interface, error handling, memory management, and finally descriptions of the functional interface. This document assumes you are thoroughly familiar with the information contained in the DWARF Debugging Information Format document.

Type Definitions

General Description

The libdwarf.h header file contains typedefs and preprocessor definitions of types and symbolic names used to reference objects of libdwarf. The types defined by typedefs contained in libdwarf.h all use the convention of adding Dwarf_ as a prefix and can be placed in three categories:

- Scalar types: The scalar types defined in libdwarf.h are defined primarily for notational convenience and identification. Depending on the individual definition, they are interpreted as a value, a pointer, or as a flag.
- Aggregate types: Some values cannot be represented by a single scalar type; they must be represented by a collection of, or as a union of, scalar and/or aggregate types.
- Opaque types: The complete definition of these types is intentionally omitted; their use is as handles for query operations, which will yield either an instance of another opaque type to be used in another query, or an instance of a scalar or aggregate type, which is the actual result.

A complete listing of libdwarf.h can be found in “Appendix1--libdwarf.h” on page 25-22.

Scalar Types

The following scalar types are defined by libdwarf.h:

```
typedef int              Dwarf_Bool;
typedef unsigned long    Dwarf_Off;
typedef unsigned long    Dwarf_Unsigned;
typedef unsigned short   Dwarf_Half;
typedef unsigned char    Dwarf_Small;
typedef signed long      Dwarf_Signed;
typedef void*            Dwarf_Addr;
typedef void             (*Dwarf_Handler)(Dwarf_Error*error, Dwarf_Addr errarg);
```

A description of these scalar types is given in Table 25-1.

Table 25-1. Scalar Types

NAME	SIZE	ALIGN-MENT	PURPOSE
Dwarf_Bool	2|4|8	2|4|8	Boolean states
Dwarf_Off	4|8	4|8	Unsigned file offset
Dwarf_Unsigned	4|8	4|8	Unsigned large integer
Dwarf_Half	2	2	Unsigned medium integer
Dwarf_Small	1	1	Unsigned small integer
Dwarf_Signed	4|8	4|8	Signed large integer
Dwarf_Addr	4|8	4|8	Unsigned program address
Dwarf_Handler	4|8	4|8	Pointer to libdwarf error handler function

Aggregate Types

The following aggregate types are defined by libdwarf.h: Dwarf_Loc, Dwarf_Locdesc, Dwarf_Ellist, Dwarf_Bounds, and Dwarf_Block.

Location Record

The Dwarf_Loc type identifies a single atom of a location description or a location expression.

```
typedef struct {
        Dwarf_Small         lr_atom;
        Dwarf_Unsigned      lr_number;
} Dwarf_Loc;
```

The lr_atom identifies the atom corresponding to the OP_* definition in dwarf.h and it represents the operation to be performed in order to locate the item in question.

The lr_number field is the operand to be used in the calculation specified by the lr_atom field; not all atoms use this field.

Location Description

The Dwarf_Locdesc type represents an ordered list of Dwarf_Loc records used in the calculation to locate an item. Note that in many cases, the location can only be calculated at run time of the associated program.

```
typedef struct {
        Dwarf_Addr          ld_lopc;
        Dwarf_Addr          ld_hipc;
        Dwarf_Unsigned      ld_cents;
        Dwarf_Loc*          ld_s;
} Dwarf_Locdesc;
```

The ld_lopc and ld_hipc fields provide an address range for which this location descriptor is valid. Both of these fields are set to zero if the location descriptor is valid throughout the scope of the item it is associated with.

The ld_cents field contains a count of the number of Dwarf_Loc entries pointed to by the ld_s field.

The ld_s field points to an array of Dwarf_Loc records.

Element List

The Dwarf_Ellist type describes an element of an enumerated type.

```
typedef struct {
        Dwarf_Signed        el_value;
        char*               el_name;
} Dwarf_Ellist;
```

The el_value field is the value associated with the corresponding element.

The el_name field is a pointer to a NULL terminated character string giving the name of the element.

Subscript Bounds Information

The Dwarf_Bounds type describes an upper or lower bound of an array subscript.

```
typedef struct {
        Dwarf_Bool              bo_isconst;
        union {
            Dwarf_Signed        constant;
            Dwarf_Locdesc       locdesc;
        }bo_;
} Dwarf_Bounds;
```

The bo_isconst field is non-zero if the bound is a constant value; otherwise, the bound is a location description or expression, which implies that it must be calculated at run time of its associated program.

The bo_ field is a union of either a constant value or a location description that specifies the upper or lower bound of the subscript.

Data Block

The Dwarf_Block type is used to contain the value of an attribute whose form is either FORM_BLOCK2 or FORM_BLOCK4; its intended use is to deliver the value for an attribute of either of these two forms.

```
typedef struct {
        Dwarf_Unsigned      bl_len;
        Dwarf_Addr*         bl_data;
} Dwarf_Block;
```

The bl_len field contains the length in bytes of the data pointed to by the bl_data field.

The bl_data field contains a pointer to the uninterpreted data.

Opaque Types

The opaque types declared in libdwarf.h are used as descriptors for queries against dwarf information stored in various debugging sections. Each time an instance of an opaque type is returned as a result of a libdwarf operation (Dwarf_Debug excepted), it should be free'd using dwarf_dealloc() When it's no longer of use. The list of opaque types defined in libdwarf.h and their intended use is described below.

```
typedef struct Debug* Dwarf_Debug;
```

An instance of the Dwarf_Debug type is created as a result of a successful call to dwarf_init() and is used as a descriptor for subsequent access to debugging information entries and/or line number entries.

```
typedef struct Die* Dwarf_Die;
```

An instance of a Dwarf_Die type is returned from a successful call to a debugging information delivery operation and is used as a descriptor for queries about information contained in that entry.

```
typedef struct Line* Dwarf_Line;
```

An instance of a Dwarf_Line type is returned from a successful call to a line number delivery operation and is used asa descriptor for queries about information contained in line number entries.

```
typedef struct Attribute* Dwarf_Attribute;
```

An instance of a Dwarf_Attribute type is returned from a successful call to an attribute delivery operation and is used as a descriptor for queries about attribute values.

```
typedef struct Subscript* Dwarf_Subscript;
```

An instance of a Dwarf_Subscript type is returned from a successful call to dwarf_nthsubscr() and is used as a descriptor for queries about array subscripts.

```
typedef struct Type* Dwarf_Type;
```

An instance of a Dwarf_Type type is returned from a successful call to dwarf_typeof() or dwarf_subscrtype() and is used as a descriptor for queries concerning data types.

```
typedef struct Global* Dwarf_Global;
```

An instance of a Dwarf_Global type is returned from a successful call to dwarf_nextglob() and is used as a descriptor for queries concerning items in the global name space.

```
typedef struct Error* Dwarf_Error;
```

For functions which accept an error argument, an instance the Dwarf_Error type is placed in the space pointed to by this argument if supplied by the client program and an error occurred within the libdwarf function. This type is used as a descriptor for queries to obtain more information concerning the error.

Error Handling

The method for detection and disposition of error conditions that arise during access of debugging information via libdwarf is consistent across all libdwarf functions that are capable of producing an error. This section describes the method used by libdwarf in notifying client programs of error conditions.

Most functions within libdwarf accept as an argument a pointer to a Dwarf_Error descriptor where error information is stored if an error is detected by the function. Routines in the client program that provide this argument can query the Dwarf_Error descriptor to determine the nature of the error and perform appropriate processing.

A client program can also specify a function to be invoked upon detection of an error at the time the library is initialized (see dwarf_init()). When a libdwarf routine detects an error, this function is called with two arguments: a code indicating the nature of the error

and a pointer provided by the client at initialization (again see dwarf_init()). This pointer argument can be used to relay information between the error handler and other routines of the client program. A client program can specify or change both the error handling function and the pointer argument after initialization using dwarf_seterrhand() and dwarf_seterrarg().

In the case where libdwarf functions are not provided an error number parameter and no error handling function was provided at initialization, libdwarf functions terminate execution by calling abort(3C).

The following lists the processing steps taken upon detection of an error:

1. Check the error argument; if not a NULL pointer, allocate and initialize a Dwarf_Error descriptor with information describing the error, place this descriptor in the area pointed to by error, and return a value indicating an error condition.
2. If an errhand argument was provided to dwarf_init() at initialization, call errhand() passing it the error descriptor and the value of the errarg argument provided to dwarf_init(). If the error handling function returns, return a value indicating an error condition.
3. Terminate program execution by calling abort(3C).

As can be seen from the above steps, the client program can provide an error handler at initialization, and still provide an error argument to libdwarf functions when it is not desired to have the error handler invoked.

If a libdwarf function is called with invalid arguments, the behavior is undefined. In particular, supplying a NULL pointer to a libdwarf function (except where explicitly permitted), or pointers to invalid addresses or uninitialized data causes undefined behavior; there turn value in such cases is undefined, and the function may fail to invoke the caller supplied error handler or to return a meaningful error number. Implementations also may abort execution for such cases.

Values returned by libdwarf functions to indicate errors are enumerated in Table 25-2.

Table 25-2. Error Indications

SYMBOLIC NAME	VALUE	USED BY
NULL	0	Functions returning a pointer
DLV_NOCOUNT	((Dwarf_Signed)-1)	Functions returning a count
DLV_BADADDR	((Dwarf_Addr) 0)	Functions returning an address
DLV_BADOFFSET	((Dwarf_Off)0)	Functions returning an offset

It is important to note that some functions can return NULL though an error did not actually occur. For example, dwarf_nextdie() returns NULL when its die argument represents the last debugging information entry to indicate that there are no further records to be processed.

Memory Management

Several of the functions that comprise libdwarf return values that have been dynamically allocated by the library. To aid in the management of dynamic memory, the function dwarf_dealloc() is provided to free storage allocated asa result of a call to a libdwarf function. This section describes the strategy that should be taken by a client program in managing dynamic storage.

Read-only Properties

All pointers returned by or as a result of a libdwarf call should be assumed to point to read-only memory. The results are undefined for libdwarf clients that attempt to write to a region pointed to by a return value from a libdwarf call.

Storage Deallocation

For most storage allocated by libdwarf, the client can simply free the storage for reuse by calling dwarf_dealloc(), providing it with a pointer to the area and an identifier that specifies what the pointer points to. For example, to free a Dwarf_Die allocated by a call to dwarf_nextdie(), the call to dwarf_dealloc() would be:

```
dwarf_dealloc(die, DLA_DIE);
```

To free storage allocated in the form of a list of pointers, each member of the list should be deallocated, followed by deallocation of the actual list itself. The following code fragment uses an invocation of dwarf_attrlist() as an example to illustrate a technique that can be used to free storage from any libdwarf routine that returns a list:

```
Dwarf_Unsigned atcnt;
Dwarf_Attribute *atlist;

if ((atcnt = dwarf_attrlist(adie,&atlist, &error))
    != DLV_NOCOUNT) {
    for (i = 0; i< atcnt; ++i) {
        /* use atlist[i] */
        dwarf_dealloc(atlist[i], DLA_ATTR);
    }
    dwarf_dealloc(atlist, DLA_LIST);
}
```

The Dwarf_Debug returned from dwarf_init() is the only dynamic storage that cannot be free'd using dwarf_dealloc(); the function dwarf_finish() will deallocate all dynamic storage associated with an instance of a Dwarf_Debug type.

The codes that identify the storage pointed to in calls to dwarf_dealloc() are described in Table 25-3.

Table 25-3. Allocation/Deallocation Identifiers

IDENTIFIER	USED TO FREE
DLA_STRING	char*
DLA_LOC	Dwarf_Loc*
DLA_LOCDESC	Dwarf_Locdesc*
DLA_ELLIST	Dwarf_ELlist*
DLA_BOUNDS	Dwarf_Bounds*
DLA_BLOCK	Dwarf_Block*
DLA_DIE	Dwarf_Die
DLA_LINE	Dwarf_Line
DLA_LINEBUF	Dwarf_Line*
DLA_ATTR	Dwarf_Attribute
DLA_TYPE	Dwarf_Type
DLA_SUBSCR	Dwarf_Subscript
DLA_GLOBAL	Dwarf_Global
DLA_ERROR	Dwarf_Error
DLA_LIST	all other lists

Functional Interface

This section describes the functions available in the libdwarf library. Each function description includes its definition, followed by a paragraph describing the function's operation.

The functions may be categorized into nine groups: initialization operations, debugging information entry delivery operations, debugging information entry query operations, array subscript query operations, type information query operations, attribute form queries, line number operations, global name space operations, and utility operations.

The following sections describe these functions.

Initialization Operations

These functions are concerned with preparing an object file for subsequent access by the functions in libdwarf and with releasing allocated resources when access is complete.

```
Dwarf_Debug dwarf_init(
    int fd,
    Dwarf_Unsigned access,
    Dwarf_Handler errhand,
    Dwarf_Addr errarg,
    Dwarf_Error *error)
```

The function dwarf_init() returns a Dwarf_Debug descriptor that represents a handle for accessing debugging records associated with the open file descriptor fd; NULL is returned if the object does not contain debugging information or an error occurred. The access argument indicates what access is allowed for the section. Currently, only the DLC_READ parameter is valid, but once libdwarf creation routines are added to the library, DLC_RDWR and DLC_WRITE will be supported. The errhand argument is a pointer to a function that will be invoked whenever an error is detected as a result of a libdwarf operation; the errarg argument is passed as an argument to the errhand function. The file descriptor associated with the fd argument must refer to an ordinary file (i.e. not a pipe, socket, device, /proc entry, etc.), be opened with the same access permissions as specified by the access argument, and cannot be closed or used as an argument to any system calls by the client until after dwarf_finish() is called; the seek position of the file associated with fd is undefined upon return of dwarf_init(). Since dwarf_init() uses the same error handling processing as other libdwarf functions (see “Error Handling” on page 25-6), client programs will generally supply an error parameter to bypass the default actions during initialization unless the default actions are appropriate.

```
void dwarf_finish(
    Dwarf_Debug dbg)
```

The function dwarf_finish() releases all libdwarf internal resources associated with the descriptor dbg and invalidates dbg.

Debugging Information Entry Delivery Operations

These functions are concerned with accessing debugging information entries.

```
Dwarf_Die dwarf_nextdie(
    Dwarf_Debug dbg,
    Dwarf_Die die,
    Dwarf_Error *error)
```

The function dwarf_nextdie() returns the next Dwarf_Die descriptor following die or NULL if die is the last entry or an error occurred. If die is NULL, the first entry is returned.

```
Dwarf_Die dwarf_siblingof(
    Dwarf_Debug dbg,
    Dwarf_Die die,
    Dwarf_Error *error)
```

The function dwarf_sibling of() returns the Dwarf_Die descriptor of the sibling of die or NULL if die is the last entry of a sibling chain or an error occurred. If die is NULL, the first entry is returned. Note that dwarf_nextdie(dbg, NULL, &error) and dwarf_siblingof(dbg, NULL, &error) are equivalent.

```
Dwarf_Die dwarf_offdie(
    Dwarf_Debug dbg,
    Dwarf_Off offset,
    Dwarf_Error *error)
```

The function dwarf_offdie() returns the Dwarf_Die descriptor of the debugging information entry at offset in the section containing debugging information entries or NULL if offset is not the start of a valid debugging information entry.

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

The function dwarf_pcfile() returns the Dwarf_Die descriptor of the compilation unit debugging information entry that contains the address of pc; NULL is returned if no entry exists or an error occurred. Currently compilation unit debugging information entries are defined as those having a tag of: TAG_compile_unit.

```
Dwarf_Die dwarf_pcsubr(
    Dwarf_Debug dbg,
    Dwarf_Addr pc,
    Dwarf_Error *error)
```

The function dwarf_pcsubr() returns the Dwarf_Die descriptor of the subroutine debugging entry that contains the address of pc, or NULL if no entry exists or an error occurred. Currently subroutine debugging information entries are defined as those having a tag of: TAG_subroutine, TAG_inlined_subroutine,or TAG_global_subroutine.

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

The function dwarf_pcscope() returns the Dwarf_Die descriptor for the debugging information entry that represents the inner most enclosing scope containing pc, or NULL if no entry exists or an error occurred. Debugging information entries that represent a scope are those containing a low pc attribute and either a high pc or byte size attribute that deliniates a range. For example: a debugging information entry for a lexical block is considered one having a scope whereas a debugging information entry for a label is not.

```
Dwarf_Die dwarf_child(
    Dwarf_Die die,
    Dwarf_Error *error)
```

The function dwarf_child() returns the Dwarf_Die descriptor of the first child of die or NULL if die does not have any children or an error occurred. The function dwarf_siblingof() can be used with the return value of dwarf_child() to access other children of die.

Debugging Information Entry Query Operations

These queries return specific information about debugging information entries or a descriptor that can be used on subsequent queries when given a Dwarf_Die descriptor. Note that some operations are specific to debugging information entries that are represented by a Dwarf_Die descriptor of a specific type. For example, not all debugging information entries contain an attribute having a name, so consequently, a call to dwarf_name() using a Dwarf_Die descriptor that does not have a name attribute will return NULL. There are three methods that can be used:

1. Call dwarf_hasattr() to determine if the debugging information entry has the attribute of interest prior to issuing the query for information about the attribute.
2. Supply an error argument and check its value after a call to a query indicates an unsuccessful return to determine the nature of the problem.
3. Arrange to have an error handling function invoked upon detection of an error (see dwarf_init()).

```
Dwarf_Signed dwarf_childcnt(
    Dwarf_Die die,
    Dwarf_Error *error)
```

The function dwarf_childcnt() returns the number of children debugging information entries of die or DLV_NOCOUNT if an error occurred. The return value represents the number of debugging information entries that exist between die and its next sibling debugging information entry.

```
Dwarf_Half dwarf_tag(
    Dwarf_Die die,
    Dwarf_Error *error)
```

The function dwarf_tag() returns the tag of die.

```
Dwarf_Off dwarf_dieoffset(
    Dwarf_Die die,
    Dwarf_Error *error)
```

The function dwarf_dieoffset() returns the position of die in the section containing debugging information entries; DLV_BADOFFSET is returned on error.

```
char* dwarf_diename(
    Dwarf_Die die,
    Dwarf_Error *error)
```

The function dwarf_diename() returns a pointer to a NULL terminated string of characters that represents the name of die; NULL is returned if die does not have a name attribute or an error occurred. The storage pointed to by a successful return of dwarf_diename() should be free'd when no longer of interest (see dwarf_dealloc()).

```
Dwarf_Bool dwarf_hasattr(
    Dwarf_Die die,
    Dwarf_Half attr,
    Dwarf_Error *error)
```

The function dwarf_hasattr() returns non-zero if die has the attribute attr and zero otherwise.

```
Dwarf_Attribute dwarf_attr(
    Dwarf_Die die,
    Dwarf_Half attr,
    Dwarf_Error *error)
```

The function dwarf_attr() returns an Dwarf_Attribute descriptor of die having the attribute name attr if die represents a debugging information entry with that attribute; NULL is returned if attr is not contained in die or an error occurred.

```
Dwarf_Type dwarf_typeof(
    Dwarf_Die die,
    Dwarf_Error *error)
```

The function dwarf_typeof() returns a Dwarf_Type descriptor that describes the type of die; NULL is returned if die does not contain a type attribute or an error occurred. In the case where die represents an array type debugging information entry, the Dwarf_Type descriptor returned by dwarf_typeof() applies to the element type of the array.

```
Dwarf_Signed dwarf_loclist(
    Dwarf_Die die,
    Dwarf_Locdesc **llbuf,
    Dwarf_Error *error)
```

The function dwarf_loclist() sets llbuf to point atan array of Dwarf_Locdesc pointers and returns the number of elements in the array; DLV_NOCOUNT is returned on error. The storage pointed to by llbuf after a successful return of dwarf_loclist() should be free'd when no longer of interest (see dwarf_dealloc()).

```
Dwarf_Locdesc* dwarf_stringlen(
    Dwarf_Die die,
    Dwarf_Error *error)
```

The function dwarf_stringlen() returns a pointer to a Dwarf_Locdesc that when evaluated, yields the length of the string represented by die; NULL is returned if die does not contain a string length attribute or an error occurred. The storage pointed to by a successful return of dwarf_stringlen() should be free'd when no longer of interest (see dwarf_dealloc()).

```
Dwarf_Signed dwarf_subscrcnt(
    Dwarf_Die die,
    Dwarf_Error *error)
```

The function dwarf_subscrcnt() returns the number of subscript attributes that are owned by the array type represented by die; DLV_NOCOUNT is returned on error.

```
Dwarf_Subscript dwarf_nthsubscr(
    Dwarf_Die die,
    Dwarf_Unsigned ssndx,
    Dwarf_Error *error)
```

The function dwarf_nthsubscr() returns a Dwarf_Subscript descriptor that represents the ssndx member of the array type debugging information entry represented by die where 1 is the first member; NULL is returned if die does not have an ssndx member or an error occurred.

```
Dwarf_Addr dwarf_lowpc(
    Dwarf_Die die,
    Dwarf_Error *error)
```

The function dwarf_lowpc() returns the low program counter value associated with the die descriptor if die represents a debugging information entry having this attribute; DLV_BADADDR is returned if die does not have this attribute or an error occurred.

```
Dwarf_Addr dwarf_highpc(
    Dwarf_Die die,
    Dwarf_Error *error)
```

The function dwarf_highpc() returns the high program counter value associated with the die descriptor if die represents a debugging information entry having this attribute; DLV_BADADDR is returned if die does not have this attribute or an error occurred.

```
Dwarf_Signed dwarf_elemlist(
    Dwarf_Die die,
    Dwarf_Ellist** elbuf,
    Dwarf_Error *error)
```

The function dwarf_elemlist() sets elbuf to point at an array of Dwarf_Ellist pointers and returns the number of elements in the array; DLV_NOCOUNT is returned on error. The storage pointed to by elbuf after a successful return of dwarf_elemlist() should be free'd when no longer of interest (see dwarf_dealloc()).

```
Dwarf_Signed dwarf_bytesize(
    Dwarf_Die die,
    Dwarf_Error *error)
```

The function dwarf_bytesize() returns the number of bytes needed to contain an instance of the aggregate debugging information entry represented by die; -1 is returned if die does not contain a byte size attribute or an error occurred.

```
Dwarf_Bool dwarf_isbitfield(
    Dwarf_Die die,
    Dwarf_Error *error)
```

The function dwarf_isbitfield() returns non-zero if die is a descriptor for a debugging information entry that represents a bit field member; zero is returned if die is not associated with a bit field member.

```
Dwarf_Signed dwarf_bitsize(
    Dwarf_Die die,
    Dwarf_Error *error)
```

The function dwarf_bitsize() returns the number of bits occupied by the bit field value; -1 is returned if die does not contain a bit size attribute or an error occurred.

```
Dwarf_Signed dwarf_bitoffset(
    Dwarf_Die die,
    Dwarf_Error *error)
```

The function dwarf_bitoffset() returns the number of bits to the left of the most significant bit of the bit field value; -1 is returned if die does not contain a bit offset attribute or an error occurred.

```
Dwarf_Signed dwarf_srclang(
    Dwarf_Die die,
    Dwarf_Error *error)
```

The function dwarf_srclang() returns a code indicating the source language of the compilation unit represented by the descriptor die; -1 is returned if die does not represent a source file debugging information entry or an error occurred.

```
Dwarf_Signed dwarf_arrayorder(
    Dwarf_Die die,
    Dwarf_Error *error)
```

The function dwarf_arrayorder() returns a code indicating the ordering of the array represented by the descriptor die; if die represents an array without an ordering attribute, the code indicating row major is returned; -1 is returned if die does not represent an array type debugging information entry or an error occurred.

```
Dwarf_Signed dwarf_attrlist(
    Dwarf_Die die,
    Dwarf_Attribute** attrbuf,
    Dwarf_Error *error)
```

The function dwarf_attrlist() sets attrbuf to point at an array of Dwarf_Attribute descriptor and returns the number of elements in the array; DLV_NOCOUNT is returned on error. The storage pointed to by attrbuf after a successful return of dwarf_attrlist() should be free'd when no longer of interest (see dwarf_dealloc()).

Array Subscript Query Operations

These operations return information concerning array subscripts.

```
Dwarf_Type dwarf_subscrtype(
    Dwarf_Subscript ss,
    Dwarf_Error *error)
```

The function dwarf_subscrtype() returns a Dwarf_Type descriptor that represents the type information for the subscript element represented by the Dwarf_Subscript descriptor ss. NULL is returned on error.

```
Dwarf_Bounds* dwarf_lobounds(
    Dwarf_Subscript ss,
    Dwarf_Error *error)
```

The function dwarf_lobounds() returns a pointer to a Dwarf_Bounds structure that describes the lower bound of the array subscript represented by the Dwarf_Subscript descriptor ss; NULL is returned on error. The storage pointed to by a successful return of dwarf_lobounds() should be free'd when no longer of interest (see dwarf_dealloc()).

```
Dwarf_Bounds* dwarf_hibounds(
    Dwarf_Subscriptss,
    Dwarf_Error *error)
```

The function dwarf_hibounds() returns a pointer to a Dwarf_Bounds structure that describes the upper bound of the array subscript represented by the Dwarf_Subscript descriptor; NULL is returned on error. The storage pointed to by a successful return of dwarf_hibounds() should be free'd when no longer of interest (see dwarf_dealloc()).

Type Information Query Operations

These operations return information concerning data types.

```
Dwarf_Signed dwarf_modlist(
    Dwarf_Type typ,
    Dwarf_Small** modbuf,
    Dwarf_Error *error)
```

The function dwarf_modlist() sets modbuf to point to an array of type modifiers represented by the Dwarf_Type descriptor typ and returns the number of elements in the array; DLV_NOCOUNT is returned on error. The storage pointed to by modbuf after a successful return of dwarf_modlist() should be free'd when no longer of interest (see dwarf_dealloc()).

```
Dwarf_Bool dwarf_isfundtype(
    Dwarf_Type typ,
    Dwarf_Error *error)
```

The function dwarf_isfundtype() returns non-zero if the Dwarf_Type descriptor typ represents a fundamental type; zero is returned otherwise.

```
Dwarf_Half dwarf_fundtype(
    Dwarf_Type typ,
    Dwarf_Error *error)
```

The function dwarf_fundtype() returns a code that indicates the fundamental type of the type represented by the descriptor typ; zero is returned if typ does not represent a type that is fundamental or an error occurred.

```
Dwarf_Die dwarf_udtype(
    Dwarf_Type udt,
    Dwarf_Error *error)
```

The function dwarf_udtype() returns a Dwarf_Die descriptor that represents the debugging information entry for the user defined type represented by the descriptor udt; NULL is returned if typ does not represent a type that is user defined or an error occurred.

Attribute Form Queries

Based on the attribute's form, these operations are concerned with returning uninterpreted attribute data. For compatibility with future DWARF versions, these functions mask off the attribute form from the name in deciding what attribute is intended. This applies to all Attribute Form Queries with the exception of dwarf_hasform(). Since it is not always obvious from the return value of these functions if an error occurred or not, one should

always supply an error parameter or have arranged to have an error handling function invoked (see dwarf_init()) to determine the validity of the return and the nature of any errors that may have occurred.

```
Dwarf_Half dwarf_atname(
    Dwarf_Attributeattr,
    Dwarf_Error *error)
```

The function dwarf_atname() returns the attribute name of the attribute represented by the Dwarf_Attribute descriptor attr. A zero is returned on error.

```
Dwarf_Bool dwarf_hasform(
    Dwarf_Attributeattr,
    Dwarf_Half form,
    Dwarf_Error *error)
```

The function dwarf_hasform() returns non-zero if the attribute represented by the Dwarf_Attribute descriptor attr has the data format of form. A zero is returned otherwise.

```
Dwarf_Off dwarf_formref(
    Dwarf_Attributeattr,
    Dwarf_Error *error)
```

The function dwarf_formref() returns the reference value of the attribute represented by the descriptor attr.

```
Dwarf_Addr dwarf_formaddr(
    Dwarf_Attributeattr,
    Dwarf_Error *error)
```

The function dwarf_formaddr() returns the address value of the attribute represented by the descriptor attr.

```
Dwarf_Unsigned dwarf_formudata(
    Dwarf_Attributeattr,
    Dwarf_Error *error)
```

The function dwarf_formudata() returns a Dwarf_Unsigned value of the attribute represented by the descriptor attr. This can be used for attributes having the form of either FORM_DATA2 or FORM_DATA4 and also FORM_DATA8 for machines supporting Dwarf_Unsigned types of 8 bytes or larger.

```
Dwarf_Signed dwarf_formsdata(
    Dwarf_Attributeattr,
    Dwarf_Error *error)
```

The function dwarf_formsdata() returns a Dwarf_Signed value of the attribute represented by the descriptor attr. This can be used or attributes having the form of either FORM_DATA2 or FORM_DATA4 and also FORM_DATA8 for machines supporting Dwarf_Signed types of 8 bytes or larger. If the size of the data attribute referenced is smaller than the size of the Dwarf_Signed type, its value is sign extended.

```
Dwarf_Block* dwarf_formblock(
    Dwarf_Attributeattr,
    Dwarf_Error *error)
```

The function dwarf_formblock() returns a pointer to a Dwarf_Block structure containing the block value of the attribute represented by the descriptor attr. This can be used for attributes having the form of either FORM_BLOCK2or FORM_BLOCK4. The storage pointed to by a successful return of dwarf_formblock() should be free'd when no longer of interest (see dwarf_dealloc()).

```
char* dwarf_formstring(
    Dwarf_Attributeattr,
    Dwarf_Error *error)
```

The function dwarf_formstring() returns a pointer to a null-terminated string containing the string value of the attribute represented by the descriptor attr. The storage pointed to by a successful return of dwarf_formstring() should be free'd when no longer of interest (see dwarf_dealloc()).

Line Number Operations

These functions are concerned with accessing line number entries, mapping debugging information entry objects to their corresponding source lines, and providing a mechanism for obtaining information about line number entries.

```
Dwarf_Line dwarf_nextline(
    Dwarf_Debug dbg,
    Dwarf_Line line,
    Dwarf_Error *error)
```

The function dwarf_nextline() returns the next line number descriptor following line or NULL if line is the last entry or an error occurred. If line is NULL, the first entry is returned.

```
Dwarf_Line dwarf_prevline(
    Dwarf_Debug dbg,
    Dwarf_Line line,
    Dwarf_Error *error)
```

The function dwarf_prevline() returns the line number descriptor preceding line or NULL if line is the first entry or an error occurred. If line is NULL, the first entry is returned.

```
Dwarf_Signed dwarf_pclines(
    Dwarf_Debug dbg,
    Dwarf_Addr pc,
    Dwarf_Line **linebuf,
    Dwarf_Signed slide,
    Dwarf_Error *error)
```

The function dwarf_pclines() places all line number descriptor that correspond to the value of pc into a single block and sets linebuf to point to that block; a count of the number of Dwarf_Line descriptor that are in this block is returned. For most cases, the count returned will be one, though this count may be higher if optimizations such as common subexpression elimination result in multiple line number entries for a given value of pc. The slide argument specifies the direction to search for the nearest line number entry in the event that there is no line number entry that contains an exact match for pc. This argument may be one of: DLS_BACKWARD, DLS_NOSLIDE, DLS_FORWARD.

DLV_NOCOUNT is returned on error. Each entry in the block pointed to by a successful return of dwarf_pc lines should be free'd using dwarf_dealloc() when no longer of interest.

```
Dwarf_Line dwarf_dieline(
    Dwarf_Die die,
    Dwarf_Error *error)
```

The function dwarf_dieline() returns the line number descriptor that corresponds to the low pc value of die or NULL if die does not contain a low pc attribute or an error occurred.

```
Dwarf_Signed dwarf_srclines(
    Dwarf_Die die,
    Dwarf_Line **linebuf,
    Dwarf_Error *error)
```

The function dwarf_srclines() places all line number descriptor for a single compilation unit into a single block, sets linebuf to point to that block, and returns the number of descriptor in this block; DLV_NOCOUNT is returned on error. The die argument must represent a debugging information entry for a compilation unit. Each entry in the block pointed to by a successful return of dwarf_srclines should be free'd using dwarf_dealloc() when no longer of interest.

```
Dwarf_Bool dwarf_is1stline(
    Dwarf_Line line,
    Dwarf_Error *error)
```

The function dwarf_is1stline() returns non-zero if line represents a line number entry that is the first of a block of line number entries for a given compilation unit. A non-zero return from dwarf_is1stline() implies that a call to dwarf_lineaddr() giving line as a descriptor will return an address that represents the base address for the source file.

```
Dwarf_Unsigned dwarf_lineno(
    Dwarf_Line line,
    Dwarf_Error *error)
```

The function dwarf_lineno() returns the source statement line number corresponding to the descriptor line.

```
Dwarf_Addr dwarf_lineaddr(
    Dwarf_Line line,
    Dwarf_Error *error)
```

The function dwarf_lineaddr() returns the address associated with the descriptor line.

```
Dwarf_Signed dwarf_lineoff(
    Dwarf_Line line,
    Dwarf_Error *error)
```

The function dwarf_lineoff() returns the off set in bytes from the beginning of the line in which the statement appears. If the generator of line number information represents statements in terms of source lines only, a-1isreturned.

```
char* dwarf_linesrc(
    Dwarf_Line line,
    Dwarf_Error *error)
```

The function dwarf_linesrc() returns a pointer to a NULL terminated string of characters that represents the name of the compilation unit where line appears; NULL is returned on error. The storage pointed to by a successful return of dwarf_linesrc() should be free'd when no longer of interest (see dwarf_dealloc()).

Global Name Space Operations

```
Dwarf_Global dwarf_nextglob(
    Dwarf_Debug dbg,
    Dwarf_Global glob,
    Dwarf_Error *error)
```

The function dwarf_nextglob() returns the next Dwarf_Global descriptor representing the next global entry following glob; NULL is returned if glob is the last global entry or an error occurred. If glob is NULL, the first global entry is returned. A global entry is currently defined as an entry that is associated with a debugging information entry having a d_tag value of: TAG_global_variable or TAG_global_subroutine.

```
char* dwarf_globname(
    Dwarf_Global glob,
    Dwarf_Error *error)
```

The function dwarf_globname() returns a pointer to a NULL terminated string of characters that represents the name of glob; NULL is returned on error. The storage pointed to by a successful return of dwarf_globname() should be free'd when no longer of interest (see dwarf_dealloc()).

```
Dwarf_Die dwarf_globdie(
    Dwarf_Global glob,
    Dwarf_Error *error)
```

The function dwarf_globdie() returns the Dwarf_Die descriptor of the debugging information entry associated with the global entry glob; NULL is returned on error.

Utility Operations

These functions aid with the management of errors encountered when using functions in the libdwarf library and releasing memory allocated as a result of a libdwarf operation.

```
Dwarf_Unsigned dwarf_errno(
    Dwarf_Error error)
```

dwarf_errno() returns the error number corresponding to the error specified by error.

```
const char* dwarf_errmsg(
    Dwarf_Error error)
```

The function dwarf_errmsg() returns a pointer to an error message string corresponding to the error specified by error or NULL if the error is out of bounds. Note that the string returned by dwarf_errmsg() should not be deallocated using dwarf_dealloc().

The minimum set of errors are enumerated in Table 25-4.

Table 25-4. Error Codes

SYMBOLIC NAME	DESCRIPTION
DLE_NE	No error (0)
DLE_VMM	Version of DWARF information newer than libdwarf
DLE_MAP	Memory map failure
DLE_LEE	Propagation of libelf error
DLE_NDS	No debug section
DLE_NLS	No line section
DLE_ID	Requested information not associated with descriptor
DLE_IOF	I/O failure
DLE_MAF	Memory allocation failure
DLE_IA	Invalid argument
DLE_MDE	Mangled debugging entry
DLE_MLE	Mangled line number entry
DLE_FNO	File descriptor does not refer to an open file
DLE_FNR	File is not a regular file
DLE_FWA	File is opened with wrong access
DLE_NOB	File is not an object file
DLE_MOF	Mangled object file header
DLE_LAST	Upper bound of libdwarf errors
DLE_LO_USER	Lower bound of implementation specific codes

This list of errors is not necessarily complete; additional errors might be added when functionality to create debugging information entries are added to libdwarf and by the implementors of libdwarf to describe internal errors not addressed by the above list.

```
Dwarf_Handler dwarf_seterrhand(
    Dwarf_Debug dbg,
    Dwarf_Handler errhand)
```

The function dwarf_seterrhand() replaces the error handler (see dwarf_init()) with errhand; the old error handler is returned.

```
Dwarf_Addr dwarf_seterrarg(
    Dwarf_Debug dbg,
    Dwarf_Addr errarg)
```

The function dwarf_seterrarg() replaces the pointer to the error handler communication area (see dwarf_init()) with errarg; a pointer to the old area is returned.

```
void dwarf_dealloc(
    void* space,
    Dwarf_Unsigned typ)
```

The function dwarf_dealloc frees all dynamic storage allocated to area pointed to by space. The argument typ. is an integer code that specifies the type pointed to by the space argument. Refer to “Memory Management” on page 25-8 for details on libdwarf memory management.

Appendix1--libdwarf.h

```
#ifndef _LIBDWARF_H
#define _LIBDWARF_H
typedef int             Dwarf_Bool;         /* boolean type*/
typedef unsigned long   Dwarf_Off;          /* 4 or8 byte file offset */
typedef unsigned long    Dwarf_Unsigned;     /* 4 or8 byte unsigned value */
typedef unsigned short  Dwarf_Half;         /* 2 byte unsigned value */
typedef unsigned char   Dwarf_Small;        /* 1 byte unsigned value */
typedef signed long     Dwarf_Signed;       /* 4 or8 byte signed value */
typedef void*           Dwarf_Addr;         /* memory address */
/* uninterpreted block of data
*/
typedef struct {
     Dwarf_Unsigned  bl_len;         /*length of block */
     Dwarf_Addr      bl_data;        /*uninterpreted data */
} Dwarf_Block;
/* location record
*/
typedef struct {
     Dwarf_Small     lr_atom;        /*location operation */
     Dwarf_Unsigned  lr_number;      /*operand */
} Dwarf_Loc;
/* location description
*/
typedef struct {
     Dwarf_Addr      ld_lopc;        /*beginning ofactive range */
     Dwarf_Addr      ld_hipc;        /*end ofactive range */
     Dwarf_Half      ld_cents;       /*count oflocation records */
     Dwarf_Loc*      ld_s;           /*pointer tolist ofsame */
} Dwarf_Locdesc;
/* element list
*/
typedef struct {
     Dwarf_Signed    el_value;       /*value ofelement */
     char*           el_name;        /*name of element */
} Dwarf_Ellist;
/* subscript bounds information
*/
typedef struct {
     Dwarf_Bool      bo_isconst;
     union {
          Dwarf_Signed    constant;
```

```
          Dwarf_Locdesc*  locdesc;
     } bo_;
} Dwarf_Bounds;

/* opaque types
*/
typedef struct Debug*      Dwarf_Debug;
typedef struct Die*        Dwarf_Die;
typedef struct Line*       Dwarf_Line;
typedef struct Attribute*  Dwarf_Attribute;
typedef struct Subscript*  Dwarf_Subscript;
typedef struct Type*       Dwarf_Type;
typedef struct Global*     Dwarf_Global;
typedef struct Error*      Dwarf_Error;
/* error handler function
*/
typedef void    (*Dwarf_Handler)(Dwarf_Error error, Dwarf_Addr errarg);
/* dwarf_dealloc() typ arguments
*/
#define DLA_STRING      0x01     /* argument points to char* */
#define DLA_LOC         0x02     /* argument points to Dwarf_Loc */
#define DLA_LOCDESC     0x03     /* argument points to Dwarf_Locdesc */
#define DLA_ELLIST      0x04     /* argument points to Dwarf_Ellist */
#define DLA_BOUNDS      0x05     /* argument points to Dwarf_Bounds */
#define DLA_BLOCK       0x06     /* argument points to Dwarf_Block */
#define DLA_DEBUG       0x07     /* argument points to Dwarf_Debug */
#define DLA_DIE         0x08     /* argument points to Dwarf_Die */
#define DLA_LINE        0x09     /* argument points to Dwarf_Line */
#define DLA_ATTR        0x0a     /* argument points to Dwarf_Attribute */
#define DLA_TYPE        0x0b     /* argument points to Dwarf_Type */
#define DLA_SUBSCR      0x0c     /* argument points to Dwarf_Subscr */
#define DLA_GLOBAL      0x0d     /* argument points to Dwarf_Global */
#define DLA_ERROR       0x0e     /* argument points to Dwarf_Error */
#define DLA_LIST        0x0f     /* argument points to a list */
/* dwarf_openscn() access arguments
*/
#define DLC_READ        0        /* readonly access */
#define DLC_WRITE       1        /* write only access */
#define DLC_RDWR        2        /* read/write access */
/* dwarf_pcline() slide arguments
*/
#define DLS_BACKWARD   -1        /* slide backward tofind line*/
#define DLS_NOSLIDE     0        /* match exactly without sliding */
#define DLS_FORWARD     1        /* slide forward to findline */
/* libdwarf error numbers
*/
#define DLE_NE          0x00     /* noerror */
#define DLE_VMM         0x01     /* dwarf format/library version mismatch */
#define DLE_MAP         0x02     /* memory map failure */
#define DLE_LEE         0x03     /* libelf error */
#define DLE_NDS         0x04     /* nodebug section */
#define DLE_NLS         0x05     /* noline section */
#define DLE_ID          0x06     /* invalid descriptor for query */
#define DLE_IOF         0x07     /* I/O failure */
```

```
#define DLE_MAF         0x08     /* memory allocation failure */
#define DLE_IA          0x09     /* invalid argument */

#define DLE_MDE         0x0a     /* mangled debugging entry */
#define DLE_MLE         0x0b     /* mangled line number entry */
#define DLE_FNO         0x0c     /* filenot open */
#define DLE_FNR         0x0d     /* filenot a regular file*/
#define DLE_FWA         0x0e     /* fileopen with wrong access */
#define DLE_NOB         0x0f     /* not anobject file */
#define DLE_MOF         0x10     /* mangled object file header */
#define DLE_LAST        DLE_MOF
#define DLE_LO_USER     0x10000
/* error return values
*/
#define DLV_BADADDR  ((Dwarf_Addr) 0)  /* for functions returning address */
#define DLV_NOCOUNT  ((Dwarf_Signed)-1) /* for functions returning count */
#define DLV_BADOFFSET (  (Dwarf_Off)0)  /* for functions returning offset */
/* initialization and termination operations
*/
Dwarf_Debug dwarf_init  (
     int fd,
     Dwarf_Unsigned access,
     Dwarf_Handler errhand,
     Dwarf_Addr errarg,
     Dwarf_Error *error
);
void dwarf_finish  (
     Dwarf_Debug dbg
);
/* DIE delivery operations
*/
Dwarf_Die dwarf_nextdie  (
     Dwarf_Debug dbg,
     Dwarf_Die die,
     Dwarf_Error* error
);
Dwarf_Die dwarf_siblingof  (
     Dwarf_Debug dbg,
     Dwarf_Die die,
     Dwarf_Error* error
);
Dwarf_Die dwarf_offdie  (
     Dwarf_Debug dbg,
     Dwarf_Off offset,
     Dwarf_Error* error
);
Dwarf_Die dwarf_pcfile  (
     Dwarf_Debug dbg,
     Dwarf_Addr pc,
     Dwarf_Error* error
);
Dwarf_Die dwarf_pcsubr  (
     Dwarf_Debug dbg,
     Dwarf_Addr pc,
```

```
      Dwarf_Error* error
);
Dwarf_Die dwarf_pcscope  (
      Dwarf_Debug dbg,
      Dwarf_Addr pc,
      Dwarf_Error* error
);
Dwarf_Die dwarf_child  (
      Dwarf_Die die,
      Dwarf_Error* error
);
/* query operations for DIEs
*/
Dwarf_Signed dwarf_childcnt  (
      Dwarf_Die die,
      Dwarf_Error* error
);
Dwarf_Half dwarf_tag  (
      Dwarf_Die die,
      Dwarf_Error* error
);
Dwarf_Off dwarf_dieoffset  (
      Dwarf_Die die,
      Dwarf_Error* error
);
Dwarf_Attribute dwarf_attr  (
      Dwarf_Die die,
      Dwarf_Half attr,
      Dwarf_Error* error
);
char* dwarf_diename  (
      Dwarf_Die die,
      Dwarf_Error* error
);
Dwarf_Bool dwarf_hasattr  (
      Dwarf_Die die,
      Dwarf_Half attr,
      Dwarf_Error* error
);
Dwarf_Type dwarf_typeof  (
      Dwarf_Die die,
      Dwarf_Error* error
);
Dwarf_Signed dwarf_loclist  (
      Dwarf_Die die,
      Dwarf_Locdesc **llbuf,
      Dwarf_Error* error
);
Dwarf_Locdesc* dwarf_stringlen  (
      Dwarf_Die die,
      Dwarf_Error *error
);
Dwarf_Signed dwarf_subscrcnt  (
      Dwarf_Die die,
```

```
     Dwarf_Error* error
);
Dwarf_Subscript dwarf_nthsubscr  (
     Dwarf_Die die,
     Dwarf_Unsigned ssndx,
     Dwarf_Error* error
);
Dwarf_Addr dwarf_lowpc  (
     Dwarf_Die die,
     Dwarf_Error* error
);
Dwarf_Addr dwarf_highpc  (
     Dwarf_Die die,
     Dwarf_Error* error
);
Dwarf_Signed dwarf_elemlist  (
     Dwarf_Die die,
     Dwarf_Ellist** elbuf,
     Dwarf_Error* error
);
Dwarf_Signed dwarf_bytesize  (
     Dwarf_Die die,
     Dwarf_Error* error
);
Dwarf_Bool dwarf_isbitfield  (
     Dwarf_Die die,
     Dwarf_Error* error
);
Dwarf_Signed dwarf_bitsize  (
     Dwarf_Die die,
     Dwarf_Error* error
);
Dwarf_Signed dwarf_bitoffset  (
     Dwarf_Die die,
     Dwarf_Error* error
);
Dwarf_Signed dwarf_srclang  (
     Dwarf_Die die,
     Dwarf_Error* error
);
Dwarf_Signed dwarf_arrayorder  (
     Dwarf_Die die,
     Dwarf_Error* error
);
Dwarf_Signed dwarf_attrlist  (
     Dwarf_Die die,
     Dwarf_Attribute** attrbuf,
     Dwarf_Error* error
);
/* query operations for subscripts
*/
Dwarf_Type dwarf_subscrtype  (
     Dwarf_Subscript ss,
     Dwarf_Error* error
```

```
);
Dwarf_Bounds* dwarf_lobounds  (
     Dwarf_Subscript ss,
     Dwarf_Error* error
);
Dwarf_Bounds* dwarf_hibounds  (
     Dwarf_Subscript ss,
     Dwarf_Error* error
);
/* query operations for types
*/
Dwarf_Signed dwarf_modlist  (
     Dwarf_Type typ,
     Dwarf_Small** modbuf,
     Dwarf_Error* error
);
Dwarf_Bool dwarf_isfundtype  (
     Dwarf_Type typ,
     Dwarf_Error* error
);
Dwarf_Half dwarf_fundtype  (
     Dwarf_Type typ,
     Dwarf_Error* error
);
Dwarf_Die dwarf_udtype  (
     Dwarf_Type udt,
     Dwarf_Error* error
);
/* query operations for attributes
*/
Dwarf_Half dwarf_atname  (
     Dwarf_Attribute attr,
     Dwarf_Error* error
);
Dwarf_Bool dwarf_hasform  (
     Dwarf_Attribute attr,
     Dwarf_Half form,
     Dwarf_Error* error
);
Dwarf_Off dwarf_formref  (
     Dwarf_Attribute attr,
     Dwarf_Error* error
);
Dwarf_Addr dwarf_formaddr  (
     Dwarf_Attribute attr,
     Dwarf_Error* error
);
Dwarf_Unsigned dwarf_formudata  (
     Dwarf_Attribute attr,
     Dwarf_Error* error
);
Dwarf_Signed dwarf_formsdata  (
     Dwarf_Attribute attr,
     Dwarf_Error* error
```

```
);
Dwarf_Block* dwarf_formblock  (
     Dwarf_Attribute attr,
     Dwarf_Error* error
);
char* dwarf_formstring  (
     Dwarf_Attribute attr,
     Dwarf_Error* error
);
/* line number operations
*/
Dwarf_Line dwarf_nextline  (
     Dwarf_Debug dbg,
     Dwarf_Line line,
     Dwarf_Error* error
);
Dwarf_Line dwarf_prevline  (
     Dwarf_Debug dbg,
     Dwarf_Line line,
     Dwarf_Error* error
);
Dwarf_Signed dwarf_pclines  (
     Dwarf_Debug dbg,
     Dwarf_Addr pc,
     Dwarf_Line **linebuf,
     Dwarf_Signed slide,
     Dwarf_Error* error
);
Dwarf_Line dwarf_dieline  (
     Dwarf_Die die,
     Dwarf_Error* error
);
Dwarf_Signed dwarf_srclines  (
     Dwarf_Die die,
     Dwarf_Line **linebuf,
     Dwarf_Error* error
);
Dwarf_Bool dwarf_is1stline  (
     Dwarf_Line line,
     Dwarf_Error* error
);
Dwarf_Unsigned dwarf_lineno  (
     Dwarf_Line line,
     Dwarf_Error* error
);
Dwarf_Addr dwarf_lineaddr  (
     Dwarf_Line line,
     Dwarf_Error* error
);
Dwarf_Signed dwarf_lineoff  (
     Dwarf_Line line,
     Dwarf_Error* error
);
char* dwarf_linesrc  (
```

```
     Dwarf_Line line,
     Dwarf_Error* error
);
/* global name space operations
*/
Dwarf_Global dwarf_nextglob  (
     Dwarf_Debug dbg,
     Dwarf_Global glob,
     Dwarf_Error *error
);
char* dwarf_globname  (
     Dwarf_Global glob,
     Dwarf_Error *error
);
Dwarf_Die dwarf_globdie  (
     Dwarf_Global glob,
     Dwarf_Error *error
);
/* utility operations
*/
Dwarf_Unsigned dwarf_errno  (
     Dwarf_Error error
);
const char* dwarf_errmsg  (
     Dwarf_Error error
);
Dwarf_Handler dwarf_seterrhand  (
     Dwarf_Debug dbg,
     Dwarf_Handler errhand
);
Dwarf_Addr dwarf_seterrarg  (
     Dwarf_Debug dbg,
     Dwarf_Addr errarg
);
void dwarf_dealloc  (
     void* space,
     Dwarf_Unsigned typ
);
#endif /*_LIBDWARF_H */
```

Index

Symbols

A

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z

Spine for 1.5” Binder

Product Name: 0.5" from top of spine, Helvetica, 36 pt, Bold

Volume Number (if any): Helvetica, 24 pt, Bold

Volume Name (if any): Helvetica, 18 pt, Bold

Manual Title(s): Helvetica, 10 pt, Bold, centered vertically within space above bar, double space between each title

Bar: 1" x 1/8" beginning 1/4" in from either side

Part Number: Helvetica, 6 pt, centered, 1/8" up