Tru64 UNIX

Programmer's Guide

Part Number: AA-RH9VD-TE

September 2002

Product Version: Tru64 UNIX Version 5.1B or higher

This manual describes the programming development environment of the Tru64 UNIX operating system, emphasizing the C programming language.

Hewlett-Packard Company
Palo Alto, California

Contents

About This Manual

1 Overview

2 The Compiler System

3 Pragma Preprocessor Directives

4 Shared Libraries

5 Debugging Programs with dbx

6 Checking C Programs with lint

7 Debugging Programs with Third Degree

8 Profiling Programs to Improve Performance

9 Using and Developing Atom Tools

10 Optimizing Techniques

11 Handling Exception Conditions

12 Developing Thread-Safe Libraries

13 OpenMP Parallel Processing

14 Posting and Receiving EVM Events

A Using 32-Bit Pointers on Tru64 UNIX Systems

B Differences in the System V Habitat

C Creating Dynamically Configurable Kernel Subsystems

D Parallel Processing — Old Style

E Handling Names of Device Special Files

F Optimizing Programs with -om and cord

Index

Examples

Figures

Tables

About This Manual

This manual describes the programming environment of the HP Tru64 UNIX operating system, with an emphasis on the C programming language. The availability of other programming languages on any system is determined by the choices made at the time the system was configured or modified.

Audience

This manual addresses all programmers who use the Tru64 UNIX operating system to create or maintain programs in any supported language.

New and Changed Features

The following major changes and additions have been made since the previous version of this manual was released:

- *Chapter 4* — *Section 4.2.4* (Options to Determine Handling of Unresolved External Symbols) now includes information about the handling of `.so` files when the `-error_unresolved` option is in effect.
- *Chapter 9* — This chapter has been heavily revised with descriptions of numerous new Atom options and routines, and two new sections have been added: *Section 9.2.3.6* (Intercepting Calls to Entry Points) and *Section 9.2.6* (Calling Application Entry Points That Have Been Replaced).
- *Chapter 14* — Two new sections have been added: *Section 14.5.1.5* (Cluster Event Data Item) and *Section 14.7.10* (Missed Events). Numerous minor clarifications and additions have been made throughout the chapter.

Organization

This manual is organized as follows:

Chapter 1 Describes the phases of program development and which programming tools to use during those phases.

Chapter 2 Describes the tools that make up the compiler system and how to use them. Topics covered include compiler commands, preprocessors, compilation options, multilanguage programs, and the archiver.

Chapter 3 Describes the implementation-specific pragmas that the C compiler supports.

Chapter 4 Describes the use, creation, and maintenance of shared libraries and discusses how symbols are resolved.

Chapter 5 Describes how to use the `dbx` debugger. It includes information about the `dbx` commands, working with the monitor, setting breakpoints, and debugging machine code.

Chapter 6 Describes how to use the `lint` command to produce clean code.

Chapter 7 Describes how to use the Third Degree tool to perform memory access checks and leak detection on an application program.

Chapter 8 Describes how to use various tools and techniques to profile your code, enabling you to find which portions of code are consuming the most execution time. It also describes how to feed profiling data back to the C compiler to provide some automatic optimization of the code.

Chapter 9 Describes how to use prepackaged Atom tools to instrument an application program for various purposes, such as to obtain profiling data or to perform cache-use analysis. It also describes how you can design and create custom Atom tools.

Chapter 10 Describes how to optimize your code using the optimizer and the `spike` postlink optimizer.

Chapter 11 Describes how to use the features of the C compiler to write a structured exception handler or a termination handler.

Chapter 12 Describes how to develop multithreaded programs.

Chapter 13 Describes some programming considerations associated with using the OpenMP parallel-processing interface.

Chapter 14 Describes how to use the Event Manager utility to post and receive event notifications.

Appendix A Describes how to use 32-bit pointers on a Tru64 UNIX system.

Appendix B Describes how to achieve source code compatibility for C language programs in the System V habitat.

Appendix C Describes how to write dynamically configurable kernel subsystems.

Appendix D Describes the old-style parallel processing pragmas implemented before OpenMP.

Appendix E Describes the routines that handle conversions between the old-style and new-style names of device special files.

Appendix F Describes how to optimize a program with the `cc` command's `-om` and `-cord` options.

Related Documents

In addition to this manual, the following manuals contain information pertaining to program development:

Programming: General

Calling Standard for Alpha Systems

Assembly Language Programmer's Guide

Programming Support Tools

Network Programmer's Guide

Compaq Portable Mathematics Library

Writing Software for the International Market

Kernel Debugging

Ladebug Debugger Manual

Writing Kernel Modules

Alpha Architecture Reference Manual, 2nd Edition (Butterworth-Hinemann Press, ISBN:1–55558–145–5)

Programming: Realtime

Guide to Realtime Programming

Programming: Streams

Programmer's Guide: STREAMS

Programming: Multithreaded Applications

Guide to the POSIX Threads Library

OpenMP C and C++ Application Programming Interface specification, available on the Internet from `http://www.openmp.org/specs/`.

General User Information

Release Notes

Icons on Tru64 UNIX Printed Manuals

The printed version of the Tru64 UNIX documentation uses letter icons on the spines of the manuals to help specific audiences quickly find the manuals that meet their needs. (You can order the printed documentation from HP.) The following list describes this convention:

G	Manuals for general users
S	Manuals for system and network administrators
P	Manuals for programmers
R	Manuals for reference page users

Some manuals in the documentation help meet the needs of several audiences. For example, the information in some system manuals is also used by programmers. Keep this in mind when searching for information on specific topics.

The *Documentation Overview* provides information on all of the manuals in the Tru64 UNIX documentation set.

Reader’s Comments

HP welcomes any comments and suggestions you have on this and other Tru64 UNIX manuals.

You can send your comments in the following ways:

- Fax: 603-884-0120 Attn: UBPG Publications, ZKO3-3/Y32
- Internet electronic mail: `readers_comment@zk3.dec.com`

  A Reader’s Comment form is located on your system in the following location:

  `/usr/doc/readers_comment.txt`

Please include the following information along with your comments:

- The full title of the manual and the order number. (The order number appears on the title page of printed and PDF versions of a manual.)
- The section numbers and page numbers of the information on which you are commenting.
- The version of Tru64 UNIX that you are using.
- If known, the type of processor that is running the Tru64 UNIX software.

The Tru64 UNIX Publications group cannot respond to system problems or technical support inquiries. Please address technical questions to your local system vendor or to the appropriate HP technical support office. Information provided with the software media explains how to send problem reports to HP.

Conventions

`%`

`$`

A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bourne, Korn, and POSIX shells.

`#`

A number sign represents the superuser prompt.

`%` **`cat`**

Boldface type in interactive examples indicates typed user input.

`file`

Italic (slanted) type indicates variable values, placeholders, and function argument names.

[|]

{ | }

In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside brackets or braces indicate that you choose one item from among those listed.

. . .

In syntax definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.

`cat`(1)

A cross-reference to a reference page includes the appropriate section number in parentheses. For example, `cat`(1) indicates that you can find information on the `cat` command in Section 1 of the reference pages.

Return

In an example, a key name enclosed in a box indicates that you press that key.

Ctrl/*x*

This symbol indicates that you hold down the first named key while pressing the key or mouse button that follows the slash. In examples, this key combination is enclosed in a box (for example, Ctrl/C).

1 Overview

This chapter describes the various phases in an application development project and the Tru64 UNIX tools that you can use during each of the phases.

This chapter addresses the following topics:

- Application development phases (Section 1.1)
- Specifications and design considerations (Section 1.2)
- Major software development tools (Section 1.3)
- Source file control (Section 1.4)
- Program installation tools (Section 1.5)
- Interprocess communications (Section 1.6)

1.1 Application Development Phases

There are five major phases in application development. Table 1–1 describes these phases and the tools and features available for use in each phase.

Table 1–1: Programming Phases and Tru64 UNIX

Phase	Tools/Features
Requirements and specifications	Standards Internationalization Security
Design	Routines Coding considerations Libraries Common files
Implementation	`vi`, `ex`, `ed`, `lint`, `grep`, `cxref`, `sed`, `time`, `dbx`, `third`, `ld`, `make`, compilers, threads
Testing	`diff`, shell scripts, `pixie`, `prof`
Maintaining	`setld`, `tar`, `sccs`, `rcs`

In many instances, the Tru64 UNIX system offers more than one tool to do a job. The choices of tools and programming languages to use are left to you.

1.2 Specification and Design Considerations

When you design an application, some of your decisions depend on the nature of the application. Tru64 UNIX provides features and tools to help you create applications that can be portable, internationalized, window-oriented, or whatever is appropriate for the needs of the users of those applications.

One of the primary design considerations concerns adhering to UNIX environment standards and portability. If you want your application to run both on Tru64 UNIX systems and on other UNIX operating systems, consider limiting your design to features that adhere to X/Open portability guidelines and POSIX standards.

You might also need to design your application so that it can be used in a variety of countries. The Tru64 UNIX operating system contains internationalization tools and functions to help you write software to be used by people working in different natural languages.

Another consideration is the terminal environment in which your application will be used. If end users have workstations or window terminals, you might want to design your application to use window displays.

1.2.1 Standards

Adhering to programming standards enhances the ability to port programs and applications between hardware platforms or even operating systems. Writing programs according to portability standards makes it easy for users to move between systems without major retraining. As part of program portability, some standards include internationalization concepts.

The following are the primary standards in the UNIX programming environment:

- ANSI
- ISO
- POSIX
- X/Open

In addition to the standards in the preceding list, the OSF Application Environment Specification (AES) specifies application-level interfaces that an application must provide to support portable applications and the semantics or protocols associated with these interfaces.

Various ANSI standards apply to specific programming tools such as languages, networks and communication protocols, character coding, and database systems. Information on conformance and extensions to a particular ANSI standard appears in the documentation set for the particular language, network system, or database system. For information about compiling C programs to adhere to ANSI standards, see Chapter 2.

The Tru64 UNIX system allows you to write programs that conform to POSIX and X/Open standards. Information on the POSIX standard is contained in *POSIX — Part 1: System Application Program Interface (API) [C Language]* for IEEE Std. 1003.1c-1994. The Tru64 UNIX header files contain POSIX and X/Open conformant information.

1.2.2 Internationalization

An internationalized application provides a run-time interface that allows users to work in their own language with culturally appropriate representations of data. The Tru64 UNIX operating system provides interfaces and utilities for you to develop internationalized applications that conform to Issue 4 of the X/Open CAE specifications. It also supports the Multibyte Support Extension (MSE) of ISO C that is part of Issue 5 of the X/Open CAE specifications.

Considerations for developing internationalized applications include:

- Language
- Cultural data
- Character sets
- Localization

To meet these considerations, your applications must not make any assumptions about language, local customs, or coded character sets. Data specific to a culture is held separate from the application's logic. You use run-time facilities to bind your application to the appropriate language message text.

For details about the Tru64 UNIX internationalization package, see *Writing Software for the International Market*.

1.2.3 Window-Oriented Applications

For information on developing window-oriented applications, see the following manuals:

OSF/Motif Programmer's Guide

Common Desktop Environment: Programmer's Guide

Common Desktop Environment: Programmer's Overview

Common Desktop Environment: Application Builder User's Guide

Common Desktop Environment: Internationalization Programmer's Guide

Common Desktop Environment: Style Guide and Certification Checklist

Common Desktop Environment: Help System Author's and Programmer's Guide

1.2.4 Secure Applications

The *Security Programming* manual provides detailed information on all aspects of creating trusted programs.

Tru64 UNIX provides a Security Integration Architecture (SIA) that allows the layering of local and distributed security authentication mechanisms onto the operating system. The SIA configuration framework isolates security-sensitive commands from the specific security mechanisms. See the Passwords appendix of the *Security Administration* manual and the `sia*`(3) reference pages for more information.

1.3 Major Software Development Tools

The Tru64 UNIX system is compatible with a number of higher-level languages, and it includes tools for linking and debugging programs.

1.3.1 Languages Supported by the Tru64 UNIX Environment

The Tru64 UNIX operating system includes an assembler (for assembly language programs) and a Java development kit (JDK). Compilers for other languages — such as C, C++, Fortran, Ada, and Pascal — are separately orderable.

For a complete list of optional products, contact your HP representative.

For more information on Java, see the Java documentation in the following directory on the system where the JDK in installed:

```
/usr/share/doclib/java/index.html
```

For more information on the assembler, see `as`(1) and the *Assembly Language Programmer's Guide*.

You can order documentation for the other languages when you order the compilers for those languages.

1.3.2 Linking Object Files

In most instances, you can use the C compiler driver command (`cc`) to link separate object files into a single executable object file.

As part of the compilation process, most compiler drivers call the linker (`ld`) to combine one or more object files into a single executable object file.

In addition, the linker resolves external references, searches libraries, and performs all other processing required to create an executable object file.

The development environment allows you to create applications composed of source code files written in different languages. In these instances, you compile each of the files separately and then link the compiled object files together in a separate step. You invoke the linker separately from the compiler by entering the `ld` command.

You can create shared libraries by using compiler driver commands or the `ld` command. In addition, you can create archive (static) libraries by using the `ar` command. For more information on how to create libraries, see Chapter 4. For detailed information on compiling and linking programs, see Chapter 2 and Chapter 4, as well as the documentation sets for the individual languages.

1.3.3 Debugging and Program Analysis Tools

The following tools are the primary debugging and program analysis tools on the Tru64 UNIX operating system:

- The `dbx` debugger (see Chapter 5 or `dbx`(1) for details)
- Program profiling tools (see Chapter 8 or `hiprof`(1), `pixie`(1), `uprofile`(1), `gprof`(1), and `prof`(1) for details)
- The Third Degree tool (see Chapter 7 or `third`(1) for details)
- The `lint` utility (see Chapter 6 or `lint`(1) for details)

The `ladebug` debugger is also supported on the Tru64 UNIX operating system. In addition to providing the features provided by the `dbx` debugger, it supports features for debugging multithreaded programs. For information on the `ladebug` debugger, which supports C, C++, and Fortran, see the *Ladebug Debugger Manual* and `ladebug`(1).

1.4 Source File Control

An integral part of creating a software application is managing the development and maintenance processes. The Tru64 UNIX operating system provides the Source Code Control System (SCCS) utility and the RCS code management system to help you store application modules in a directory, track changes made to those module files, and monitor user access to the files.

SCCS and RCS on the Tru64 UNIX operating system provide support similar to SCCS and RCS utilities on other UNIX systems. In addition, Tru64 UNIX has an `sccs` preprocessor, which provides an interface to the more traditional SCCS commands.

SCCS and RCS maintain a record of changes made to files stored using the utility. The record can include information on why the changes were made, who made them, and when they were made. You can use either SCCS or RCS to recover previous versions of files as well as to maintain different versions simultaneously. SCCS is useful for application project management because it does not allow two people to modify the same file simultaneously.

For more information , see `sccs`(1), `rcs`(1), and the *Programming Support Tools* manual.

1.5 Program Installation Tools

After you create your program or application, you might want to package it as a kit for the `setld` installation utility so that it can be easily distributed to other users. The Tru64 UNIX operating system has several utilities that you can use to install, remove, combine, validate, and configure programs and applications.

Software for Tru64 UNIX systems consists of a hierarchical group of files and directories. If your application or program consists of more than one file or directory, you need to determine how the files and directories are grouped within the hierarchy. The `setld` installation process preserves the integrity of each product's hierarchy when it is transferred from the development system to a production system (that is, when the product is installed). The kitting process includes grouping the component files for the product into subsets, allowing the system administrator to install some or all of them as needed.

Using the `setld` utility and its related tools provides the following benefits:

- Installation security

  The `setld` utility verifies each subset immediately after it is transferred from one system to another to make sure that the transfer was successful. Each subset is recoverable, so you can reinstall one that has been damaged or deleted.

- Flexibility

  System administrators can choose which optional subsets to install. Administrators can also delete subsets and then reinstall them later, as needed. You might use this feature to provide multiple language support for your application or to allow users to select among optional features of your application.

- Uniformity

  The `setld` utility is an integral part of the Tru64 UNIX installation implementation.

Using `setld`, you can load your application on any of the following distribution media for installation on other systems:

- CD-ROM distribution media
- An arbitrary, mountable file system on any supported data disk; for example, a third-party SCSI disk cartridge

For more information on using the `setld` command and creating and managing software product kits, see the *Programming Support Tools* manual.

1.6 Overview of Interprocess Communication Facilities

Interprocess communication (IPC) is the exchange of information between two or more processes. In single-process programming, modules within a single process communicate with each other using global variables and function calls, with data passing between the functions and the callers. When programming using separate processes having images in separate address spaces, you need to use additional communication mechanisms.

Tru64 UNIX provides the following facilities for interprocess communication:

- System V IPC

  System V IPC includes the following IPC facilities: messages, shared memory, and semaphores. For information, see System V manuals and `mq_open`(3), `shmctl`(2), and `sem_open`(3).

- FIFO special files (named pipes)

  For information about FIFO special files, see `mkfifo`(3) and `mknod`(8).

- Signals

  For information about signals, see the *Guide to Realtime Programming*.

- Sockets

  For information about sockets, see the *Network Programmer's Guide*.

- STREAMS

  For information about STREAMS, see the *Programmer's Guide: STREAMS*.

- Threads

  For information about programming using threads, see the *Guide to the POSIX Threads Library* and Chapter 12.

- X/Open Transport Interface (XTI)

  For information about XTI, see the *Network Programmer's Guide*.

2

The Compiler System

This chapter contains information on the following topics:

- Compiler system components (Section 2.1)
- Data types in the Tru64 UNIX environment (Section 2.2)
- Using the C preprocessor (Section 2.3)
- Compiling source programs (Section 2.4)
- Linking object files (Section 2.5)
- Running programs (Section 2.6)
- Object file tools (Section 2.7)
- ANSI name space pollution cleanup in the standard C library (Section 2.8)
- Inline assembly code-ASMs (Section 2.9)

The compiler system is responsible for converting source code into an executable program. This can involve several steps:

- Preprocessing — The compiler system performs such operations as expanding macro definitions or including header files in the source code.
- Compiling — The compiler system converts a source file or preprocessed file to an object file with the `.o` file suffix.
- Linking — The compiler system produces a binary image.

These steps can be performed by separate preprocessing, compiling, and linking commands, or they can be performed in a single operation, with the compiler system calling each tool at the appropriate time during the compilation.

Other tools in the compiler system help debug the program after it has been compiled and linked, examine the object files that are produced, create libraries of routines, or analyze the run-time performance of the program.

Table 2–1 summarizes the tools in the compiler system and points to the chapter or section where they are described in this and other documents.

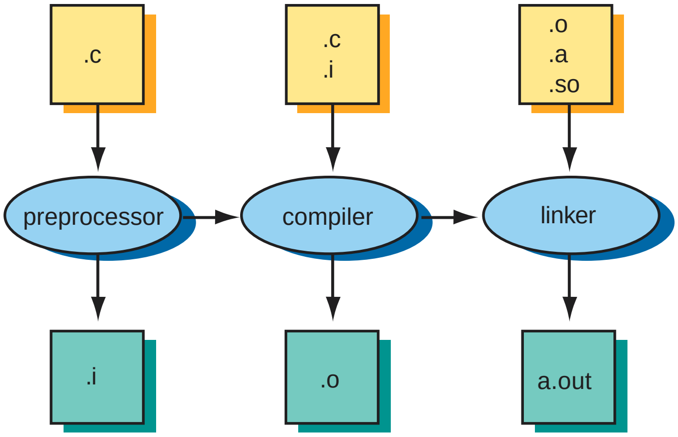
Table 2–1: Compiler System Functions

Task	Tools	Where Documented
Compile, link, and load programs; build shared libraries	Compiler drivers, link editor, dynamic loader	This chapter, Chapter 4, cc(1), c89(1), as(1), ld(1), loader(5), *Assembly Language Programmer's Guide*, *Compaq C Language Reference Manual*
Debug programs	Symbolic debugger (dbx and ladebug) and Third Degree	Chapter 5, Chapter 6, Chapter 7, dbx(1), third(1), ladebug(1), *Ladebug Debugger Manual*
Profile programs	Profiler, call graph profiler	Chapter 8, hiprof(1), pixie(1), gprof(1), prof(1), atom(1)
Optimize programs	Optimizer, postlink optimizer	This chapter, Chapter 10, cc(1), Chapter 7, third(1)
Examine object files	nm, file, size, dis, odump, and stdump tools	This chapter, nm(1), file(1), size(1), dis(1), odump(1), stdump(1), *Programming Support Tools*
Produce necessary libraries	Archiver (ar), linker (ld) command	This chapter, Chapter 4, ar(1), ld(1)

2.1 Compiler System Components

Figure 2–1 shows the relationship between the major components of the compiler system and their primary inputs and outputs.

Figure 2–1: Compiling a Program



ZK-1079U-AI

Compiler system commands, sometimes called driver programs, invoke the components of the compiler system. Each language has its own set of compiler commands and options.

The `cc` command invokes the C compiler. In the Tru64 UNIX programming environment, a single `cc` compiler command can perform multiple actions, including the following:

- Determine whether to call the appropriate preprocessor, compiler (or assembler), or linker based on the file name suffix of each file. Table 2–2 lists the supported file suffixes, which identify the contents of the input files.
- Compile and link a source file to create an executable program. If multiple source files are specified, the files can be passed to other compilers before linking.
- Assemble one or more `.s` files, which are assumed to contain assembler code, by calling the `as` assembler, and link the resulting object files. (Note that if you directly invoke the assembler, you need to link the object files in a separate step; the `as` command does not automatically link assembled object files.)
- Prevent linking and the creation of the executable program, thereby retaining the `.o` object file for a subsequent link operation.
- Pass the major options associated with the link command (`ld`) to the linker. For example, you can include the `–L` option as part of the `cc` command to specify the directory path to search for a library. Each language requires different libraries at link time; the driver program for a language passes the appropriate libraries to the linker. For more information on linking with libraries, see Chapter 4 and Section 2.5.3.
- Create an executable program file with a default name of `a.out` or with a name that you specify.

Table 2–2: File Suffixes and Associated Files

Suffix	File
`.a`	Archive library.
`.c`	C source code.
`.i`	The driver assumes that the source code was processed by the C preprocessor and that the source code is that of the processing driver; for example, `% cc -c source.i`. The file, `source.i`, is assumed to contain C source code.
`.o`	Object file.
`.s`	Assembly source code.
`.so`	Shared object (shared library).

2.2 Data Types in the Tru64 UNIX Environment

The following sections describe how data items are represented on the Tru64 UNIX system.

Note

The default memory access size on a Tru64 UNIX system is 8 bytes (quadword). This means that when two or more threads of execution are concurrently modifying adjacent memory locations, those locations must be quadword aligned to protect the individual modifications from being erroneously overwritten. Errors can occur, for example, if separate data items stored within a single quadword of a composite data structure are being concurrently modified.

For details on the problems that non-quadword alignment can cause and the various situations in which the problems can occur, see the Granularity Considerations section in the *Guide to the POSIX Threads Library*.

2.2.1 Data Type Sizes

The Tru64 UNIX system is little-endian; that is, the address of a multibyte integer is the address of its least significant byte and the more significant bytes are at higher addresses. The C compiler supports only little-endian byte ordering. The following table gives the sizes of supported data types:

Data Type	Size, in Bits
`char`	8
`short`	16
`int`	32
`long`	64
`long long`	64
`float`	32 (IEEE single)
`double`	64 (IEEE double)
`pointer`	64[a]
`long double`	128

[a] 32-bit pointers available with `-xtaso_short`.

2.2.2 Floating-Point Range and Processing

The C compiler supports IEEE single-precision (32-bit `float`), double-precision (64-bit `double`), and double-extended-precision (128-bit `long double`) floating-point data, as defined by the *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754-1985).

Floating-point numbers have the following ranges:

- `float`: 1.17549435e-38f to 3.40282347e+38f
- `double`: 2.2250738585072014e-308 to 1.79769313486231570e+308
- `long double`: 3.3621031431120935062626778173217526026e-4932 to 1.1897314953572317650857593266280070162e+4932

Tru64 UNIX provides the basic floating-point number formats, operations (add, subtract, multiply, divide, square root, remainder, and compare), and conversions defined in the standard. You can obtain full IEEE-compliant trapping behavior (including NaN [not-a-number]) by specifying a compilation option, or by specifying a fast mode when IEEE-style traps are not required. You can also select, at compile time, the rounding mode applied to the results of IEEE operations. See `cc`(1) for information on the options that support IEEE floating-point processing.

A user program can control the delivery of floating-point traps to a thread by calling `ieee_set_fp_control`(), or dynamically set the IEEE rounding mode by calling `write_rnd`(). See `ieee`(3) for more information on how to handle IEEE floating-point exceptions.

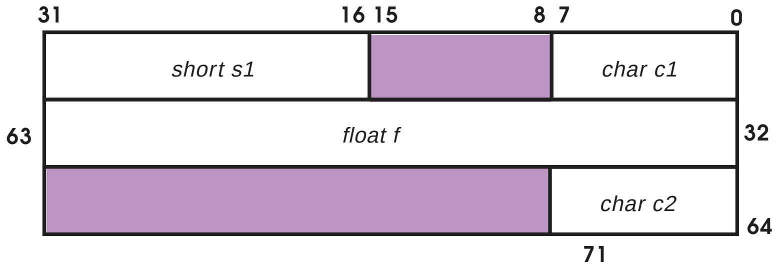
2.2.3 Structure Alignment

The C compiler aligns structure members on natural boundaries by default. That is, the components of a structure are laid out in memory in the order in which they are declared. The first component has the same address as the entire structure. Each additional component follows its predecessor on the next natural boundary for the component type.

For example, the following structure is aligned as shown in Figure 2–2:

```
struct {char c1;
        short s1;
        float f;
        char c2;
       }
```

Figure 2–2: Default Structure Alignment



ZK-1082U-AI

The first component of the structure, c1, starts at offset 0 and occupies the first byte. The second component, s1, is a short; it must start on a word boundary. Therefore, padding is added between c1 and s1. No padding is needed to make f and c2 fall on their natural boundaries. However, because size is rounded up to a multiple of f's alignment, three bytes of padding are added after c2.

You can use the following mechanisms to override the default alignment of structure members:

- The #pragma member_alignment and #pragma nomember_alignment directives
- The #pragma pack directive
- The –Zp*n* option

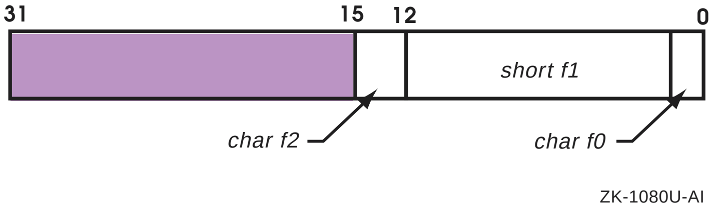
See Section 3.8 and Section 3.11 for information on these directives.

2.2.4 Bit-Field Alignment

In general, the alignment of a bit field is determined by the bit size and bit offset of the previous field. For example, the following structure is aligned as shown in Figure 2–3:

```
struct a {
    char  f0:   1;
    short f1:  12;
    char  f2:   3;
} struct_a;
```

Figure 2–3: Default Bit-Field Alignment



The first bit field, `f0`, starts on bit offset 0 and occupies 1 bit. The second, `f1`, starts at offset 1 and occupies 12 bits. The third, `f2`, starts at offset 13 and occupies 3 bits. The size of the structure is two bytes.

Certain conditions can cause padding to occur prior to the alignment of the bit field:

- Bit fields of size 0 cause padding to the next pack boundary. (The pack boundary is determined by the `#pragma pack` directive or the –`Zp`*n* compiler option.) For bit fields of size 0, the bit field's base type is ignored. For example, consider the following structure:

  ```
  struct b {
      char f0:  1;
      int    :  0;
      char f1:  2;
  } struct_b;
  ```

  If the source file is compiled with the –`Zp1` option or if a `#pragma pack 1` directive is encountered in the compilation, `f0` would start at offset 0 and occupy 1 bit, the unnamed bit field would start at offset 8 and occupy 0 bits, and `f1` would start at offset 8 and occupy 2 bits.

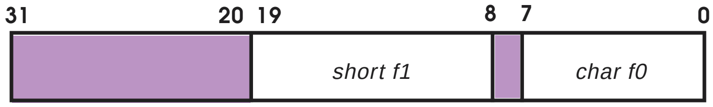
  Similarly, if the –`Zp2` option or the `#pragma pack 2` directive were used, the unnamed bit field would start at offset 16. With –`Zp4` or `#pragma pack 4`, it would start at offset 32.

- If the bit field does not fit in the current unit, padding occurs to either the next pack boundary or the next unit boundary, whichever is closest. (The unit boundary is determined by the bit field's base type; for example, the unit boundary associated with the declaration "`char foo: 1`" is a byte.) The current unit is determined by the current offset, the bit field's base size, and the kind of packing specified, as shown in the following example:

  ```
  struct c {
      char  f0:  7;
      short f1: 11;
  } struct_c;
  ```

Assuming that you specify either the –Zp1 option or the #pragma pack 1 directive, f0 starts on bit offset 0 and occupies 7 bits in the structure. Because the base size of f1 is 8 bits and the current offset is 7, f1 will not fit in the current unit. Padding is added to reach the next unit boundary or the next pack boundary, whichever comes first, in this case, bit 8. The layout of this structure is shown in Figure 2–4.

Figure 2–4: Padding to the Next Pack Boundary



ZK-1081U-AI

2.2.5 The __align Storage Class Modifier

Data alignment is implied by data type. For example, the C compiler aligns an int (32 bits) on a 4-byte boundary and a long (64 bits) on an 8-byte boundary. The __align storage-class modifier aligns objects of any of the C data types on the specified storage boundary. It can be used in a data declaration or definition.

The __align modifier has the following format:

__align (*keyword*)

__align (*n*)

Where *keyword* is a predefined alignment constant and *n* is an integer power of 2. The predefined constant or power of 2 tells the compiler the number of bytes to pad in order to align the data.

For example, to align an integer on the next quadword boundary, use any of the following declarations:

```
int __align( QUADWORD ) data;
int __align( quadword ) data;
int __align( 3 ) data;
```

In this example, int __align (3) specifies an alignment of 2x2x2 bytes, which is 8 bytes, or a quadword of memory.

The following table shows the predefined alignment constants, their equivalent power of 2, and equivalent number of bytes:

Constant	Power of 2	Number of Bytes
BYTE or byte	0	1
WORD or word	1	2
LONGWORD or longword	2	4
QUADWORD or quadword	3	8

2.3 Using the C Preprocessor

The C preprocessor performs macro expansion, includes header files, and executes preprocessor directives prior to compiling the source file. The following sections describe the Tru64 UNIX specific operations performed by the C preprocessor. For more information on the C preprocessor, see cc(1), cpp(1), and the *Compaq C Language Reference Manual*.

2.3.1 Predefined Macros

When the compiler is invoked, it defines C preprocessor macros that identify the language of the input files and the environments on which the code can run. See cc(1) for a list of the preprocessor macros. You can reference these macros in #ifdef statements to isolate code that applies to a particular language or environment. Use the following statement to uniquely identify Tru64 UNIX:

```
#if defined (__digital__) && defined (__unix__)
```

The type of source file and the type of standards you apply determine the macros that are defined. The C compiler supports several levels of standardization:

- The −std option enforces the ANSI C standard, but allows some common programming practices disallowed by the standard, and defines the macro __STDC__ to be 0 (zero). This is the default.
- The −std0 option enforces the Kernighan and Ritchie (K & R) programming style, with certain ANSI extensions in areas where the K & R behavior is undefined or ambiguous. In general, −std0 compiles most pre-ANSI C programs and produces expected results. It does not define the __STDC__ macro.
- The −std1 option strictly enforces the ANSI C standard and all of its prohibitions (such as those that apply to handling a void, the definition of an lvalue in expressions, the mixing of integrals and pointers, and the modification of an rvalue). It defines the __STDC__ macro to be 1.

Note that the `cpp` command does not predefine any macros other than the standard C macros `__LINE__`, `__FILE__`, `__DATE__`, `__TIME__`, (and `__STDC__` if appropriate).

2.3.2 Header Files

Header files are typically used for the following purposes:

- To define interfaces to system libraries
- To define constants, types, and function prototypes common to separately compiled modules in a large application

C header files, sometimes known as include files, have a `.h` suffix. Typically, the reference page for a library routine or system call indicates the required header files. Header files can be used in programs written in different languages.

Note

If you intend to debug your program using `dbx` or `ladebug`, do not place executable code in a header file. The debugger interprets a header file as one line of source code; none of the source lines in the file appears during the debugging session. For more information on the `dbx` debugger, see Chapter 5. For details on `ladebug`, see the *Ladebug Debugger Manual*.

You can include header files in a program source file in one of two ways:

`#include "`*`filename`*`"`

This indicates that the C macro preprocessor should first search for the include file *`filename`* in the directory in which it found the file that contains the directive, then in the search path indicated by the `–I` options, and finally in `/usr/include`.

`#include <`*`filename`*`>`

This indicates that the C macro preprocessor should search for the include file *`filename`* in the search path indicated by the `–I` options and then in `/usr/include`, but not in the directory where it found the file that contains the directive.

You can also use the `–I`*`dir`* and `-nocurrent_include` options to specify additional pathnames (directories) to be searched by the C preprocessor for `#include` files:

- For `-I`*`dir`*, the C preprocessor searches first in the directory where it found the file that contains the directive, followed by the specified pathname (*dir*), and then the default directory (/usr/include). If *dir* is omitted, the default directory is not searched.

- For -I with no arguments, the C preprocessor does not search in /usr/include.
- For -nocurrent_include, the C preprocessor does not search the directory containing the file that contains the #include directive; that is, #include "*filename*" is treated the same as #include <*filename*>.

2.3.3 Setting Up Multilanguage Include Files

C, Fortran, and assembly code can reside in the same include files, and can then be conditionally included in programs as required. To set up a shareable include file, you must create a .h file and enter the respective code, as shown in the following example:

```
#ifdef __LANGUAGE_C__
  .
  .    (C code)
  .
#endif
#ifdef __LANGUAGE_ASSEMBLY__
  .
  .    (assembly code)
  .
#endif
```

When the compiler includes this file in a C source file, the __LANGUAGE_C__ macro is defined and the C code is compiled. When the compiler includes this file in an assembly language source file, the __LANGUAGE_ASSEMBLY__ macro is defined, and the assembly language code is compiled.

2.3.4 Implementation-Specific Preprocessor Directives (#pragma)

The #pragma directive is a standard method of implementing features that vary from one compiler to the next. The C compiler supports the following implementation-specific pragmas:

- #pragma assert
- #pragma environment
- #pragma extern_model
- #pragma extern_prefix
- #pragma function

- #pragma inline
- #pragma intrinsic
- #pragma linkage
- #pragma member_alignment
- #pragma message
- #pragma optimize
- #pragma pack
- #pragma pointer_size
- #pragma use_linkage
- #pragma weak

Chapter 3 provides detailed descriptions of these pragmas.

2.4 Compiling Source Programs

The compilation environment established by the cc command produces object files that comply with the common object file format (COFF).

Options supported by the cc command select a variety of program development functions, including debugging, optimizing, and profiling facilities, and the names assigned to output files. See cc(1) for details on cc command-line options.

The following sections describe the default compiler behavior and how to compile multilanguage programs.

2.4.1 Default Compilation Behavior

Most compiler options have default values that are used if the option is not specified on the command line. For example, the default name for an output file is *filename*.o for object files, where *filename* is the base name of the source file. The default name for an executable program object is a.out. The following example uses the defaults in compiling two source files named prog1.c and prog2.c:

```
% cc prog1.c prog2.c
```

This command runs the C compiler, creating object files prog1.o and prog2.o and the executable program a.out.

When you enter the cc compiler command with no other options, the following options are in effect:

`-noansi_alias`

Turns off ANSI C aliasing rules, which prevents the optimizer from being aggressive in its optimizations.

`-arch generic`

Generates instructions that are appropriate for all Alpha processors.

`-assume aligned_objects`

Allows the compiler to make such an assumption, and thereby generate more efficient code for pointer dereferences of aligned pointer types.

`-assume math_errno`

Allows the compiler to make the assumption that the program might interrogate `errno` after any call to a math library routine that is capable of setting `errno`.

`-call_shared`

Produces a dynamic executable file that uses shareable objects at run time.

`-nocheck_bounds`

Disables the run-time checking of array bounds.

`-cpp`

Causes the C macro preprocessor to be called on C and assembly source files before compiling.

`-error_limit 30`

Limits the number of error-level diagnostics that the compiler will output for a given compilation to 30.

`-float`

Informs the compiler that it is not necessary to promote expressions of type `float` to type `double`.

`-nofp_reorder`

Directs the compiler not to reorder floating-point computations in a way that might affect accuracy.

`-fprm n`

Performs normal rounding (unbiased round to nearest) of floating-point numbers.

`-fptm n`

Generates instructions that do not generate floating-point underflow or inexact trapping modes.

`-g0`

Does not produce symbol information for symbolic debugging.

`-I/usr/include`

Specifies that `#include` files whose names do not begin with a slash (/) are always sought first in the directory `/usr/include`.

`-inline manual`

Inlines only those function calls explicitly requested for inlining by a `#pragma inline` directive.

`-intrinsics`

Directs the compiler to recognize certain functions as intrinsics and perform appropriate optimizations.

`-member_alignment`

Directs the compiler to naturally align data structure members (with the exception of bit-field members).

`-nomisalign`

Generates alignment faults for arbitrarily aligned addresses.

`-nestlevel=50`

Sets the nesting level limit for include files to 50.

`-newc`

Invokes the compiler with all of the default option settings listed. This option is provided only to turn off `-migrate`.

`-O1`

Enables global optimizations.

`-p0`

Disables profiling.

`-nopg`

Turns off `gprof` profiling.

`-preempt_module`

Allows symbol preemption on a module-by-module basis.

`-SD/usr/include`

Suppresses messages for nonportable constructs in header files whose pathnames are prefixed with `/usr/include`.

`-signed`

Causes type `char` to use the same representation as `signed char`.

`-std`

Enforces the ANSI C standard, but allows some common programming practices disallowed by the standard.

`-tune generic`

Selects instruction tuning that is appropriate for all implementations of the Alpha architecture.

`-writable_strings`

Makes string literals writable.

Table 2–3 lists the default options according to the option categories used in `cc`(1).

Table 2–3: cc Command Default Options, by Option Category

Option Category	Default Options
Compiler selection	`-newc`
Language mode	`-std`
Overall compiler behavior	`-arch generic`, `-error_limit 30`, `-nestlevel=50`
Compiler diagnostic controls	`-SD/usr/include`
C preprocessor	`-cpp`, `-I/usr/include`
Linker or loader	`-call_shared`

Table 2–3: cc Command Default Options, by Option Category (cont.)

Option Category	Default Options
Optimization	`-noansi_alias`, `-assume math_errno`, `-float`, `-nofp_reorder`, `-inline manual`, `-intrinsics`, `-O1`, `-preempt_symbol`, `-tune generic`
Feedback-directed optimization	None
Source-code debugging	`-g0`
Program profiling	`-p0`, `-nopg`
Data alignment	`-assume aligned_objects`, `-member_alignment`, `-nomisalign`
Data volatility	`-weak_volatile`
C language	`-signed`, `-writable_strings`
Stack-handling and pointer-handling	None
IEEE floating-point support	`-fprm n`, `-fptm n`
Compiler development	None

The following list includes miscellaneous aspects of the default `cc` compiler behavior:

- Source files are automatically linked if compilation (or assembly) is successful.
- The output file is named `a.out` unless another name is specified by using the `-o` option.
- Floating-point computations are fast floating point, not full IEEE.
- Pointers are 64 bits. For information on using 32-bit pointers, see Appendix A.
- Temporary files are placed in the `/tmp` directory or the directory specified by the environment variable `$TMPDIR`.

2.4.2 Compiling Multilanguage Programs

When the source language of the main program differs from that of a subprogram, compile each program separately with the appropriate driver and link the object files in a separate step. You can create objects suitable for linking by specifying the `-c` option, which stops a driver immediately after the object file has been created. For example:

```
% cc -c main.c
```

This command produces the object file `main.o`, not the executable file `a.out`.

After creating object modules for source files written in languages other than C, you can use the cc command to compile C source files and link all of the object modules into an executable file. For example, the following cc command compiles c-prog.c and links c-prog.o and nonc-prog.o into the executable file a.out:

```
% cc nonc-prog.o c-prog.c
```

2.4.3 Enabling Run-Time Checking of Array Bounds

The cc command's -check_bounds option generates run-time code to perform array bounds verification. The -nocheck_bounds option (the default) disables the run-time checking of array bounds.

The kind of code that causes the compiler to emit run-time checks, and the exact bounds values used in a given check, are subject to certain characteristics of the compiler implementation that would not be obvious to a user. The exact conditions, which assume a good understanding of the C language rules involving arrays, are as follows:

- Checks are made only when the name of a declared array object is used. No checks are made when a pointer value is used, even if the pointer is dereferenced using the subscript operator. This means, for example, that no checks are made on formal parameters declared as arrays of one dimension because they are considered pointers in the C language. However, if a formal parameter is a multidimensional array, the first subscript represents a pointer-manipulation that determines the array object to be accessed, and that bound cannot be checked, but bounds checks are generated for the second and subsequent subscripts.
- If an array is accessed using the subscript operator (as either the left or right operand) and the subscript operator is not the operand of an address-of operator, the check is for whether the index is between zero and the number of elements in the array minus one inclusive.
- If an array is accessed using the subscript operator (as either the left or right operand) and the subscript operator *is* the operand of the address-of operator, the check is for whether the index is between zero and the number of elements in the array inclusive. The C language specifically requires that it be valid to use the address that is one past the end of an array in a computation, to allow such common programming practice as loop termination tests like:

  ```
  int a[10];
  int *b;
  for (b = a ; b < &a[10] ; b++) { .... }
  ```

  In this case, the use of &a[10] is allowed even though a[10] is outside the bounds of the array.

- If the array is being accessed using pointer addition, the check is for whether the value being added is between zero and the number of elements in the array inclusive. Adding an integer to an array name involves converting the array name to a pointer to the first element and adding the integer value scaled by the size of an element. The implementation of bounds checking in the compiler is triggered by the conversion of the array name to a pointer, but at the point in time when the bounds check is introduced, it is not known whether the resulting pointer value will be dereferenced. Therefore, this case is treated like the previous case: only the computation of the address is checked and it is valid to compute the address of one element beyond the end of the array.
- If the array is being accessed using pointer subtraction (that is, the subraction of an integer value from a pointer, not the subtraction of one pointer from another), the check is for whether the value being subtracted is between the negation of the number of elements in the array and zero inclusive.

  Note that in the last three cases, an optional compile-time message (`ident SUBSCRBOUNDS2`) can be enabled to detect the case where an array has been accessed using either a constant subscript or constant pointer arithmetic, and the element accessed is exactly one past the end of the array.
- No check is made for arrays declared with one element. Because ANSI C does not allow arrays without dimensions inside `struct` declarations, it is common practice to implement a dynamic-sized array as a `struct` that holds the number of elements allocated in some member, and whose last member is an array declared with one element. Because accesses to the final array member are intended to be bounded by the run-time allocated size, it is not useful to check against the declared bound of 1.

  Note that in this case, an optional compile-time message (`ident SUBSCRBOUNDS1`) can be enabled to detect the case where an array declared with a single element has been accessed using either a constant subscript or constant pointer arithmetic, and the element accessed is not part of the array.
- The compiler will emit run-time checks for arrays indexed by constants (even though the compiler can and does detect this case at compile-time). An exception would be that no run-time check is made if the compiler can determine that the access is valid.
- If a multidimensional array is accessed, the compiler will perform checks on each of the subscript expressions, making sure each is within the corresponding bound. So, for the following code the compiler will check that both `x` and `y` are between 0 and 9 (it will not check that `10 * x + y` is between 0 and 99):

```
    int a[10][10];
    int x,y,z;
    x = a[x][y];
```

The following examples illustrate these rules:

```
int a[10];
int *b;
int c;
int *d;
int one[1];
int vla[c];          // C9X variable-length array

a[c] = 1;            // check c is 0-9, array subscript
c[a] = 1;            // check c is 0-9, array subscript
b[c] = 1;            // no check, b is a pointer
d = a + c;           // check c is 0-10, computing address
d = b + c;           // no check, b is a pointer
b = &a[c]            // check c is 0-10, computing address
*(a + c) = 1;        // check c is 0-10, computing address
*(a - c) = 1;        // check c is -10 to 0, computing address

a[1] = 1;            // no run-time check - know access is valid
vla[1] = 1;          // run-time check, vla has run-time bounds
a[10] = 1;           // run-time check  (and compiler diagnostic)
d = a + 10;          // no run-time check, computing address
                     // SUBSCRBOUNDS2 message can be enabled

c = one[5];          // no run-time check, array of one element
                     // SUBSCRBOUNDS1 message can be enabled
```

When an out-of-bounds access is encountered, the output is as follows:

```
Trace/BPT trap (core dumped)
```

A program can trap this error with the following code:

```
signal(SIGTRAP, handler);
```

Note that when run-time checking is enabled, incorrect checks might be made in certain cases where arrays are legitimately accessed using pointer arithmetic.

The compiler is only able to output the checking code for the first arithmetic operation performed on a pointer that results from converting an array name to a pointer. This can result in an incorrect check if the resulting pointer value is again operated on by pointer arithmetic. Consider the expression `a = b + c - d`, where `a` is a pointer, `b` is an array, and `c` and `d` are integers. When bounds-checking is enabled, a check will be made to verify that `c` is within the bounds of the array. This will lead to an incorrect run-time trap in cases where `c` is outside the bounds of the array but `c - d` is not.

In these cases, you can recode the pointer expression so that the integer part is in parentheses. This way, the expression will contain only one pointer arithmetic operation and the correct check will be made. In the previous example, the expression would be changed to the following:

```
a = b + (c - d);
```

2.5 Linking Object Files

The `cc` driver command can link object files to produce an executable program. In some cases, you may want to use the `ld` linker directly. Depending on the nature of the application, you must decide whether to compile and link separately or to compile and link with one compiler command. Factors to consider include:

- Whether all source files are in the same language
- Whether any files are in source form

2.5.1 Linking with Compiler Commands

You can use a compiler command instead of the linker command to link separate objects into one executable program. Each compiler (except the assembler) recognizes the `.o` suffix as the name of a file that contains object code suitable for linking and immediately invokes the linker.

Because the compiler driver programs pass the libraries associated with that language to the linker, using the compiler command is usually recommended. For example, the `cc` driver uses the C library (`libc.so`) by default. For information about the default libraries used by each compiler command, see the appropriate command in the reference pages, such as `cc`(1).

You can also use the `-l` option of the `cc` command to specify additional libraries to be searched for unresolved references. The following example shows how to use the `cc` driver to pass the names of two libraries to the linker with the `-l` option:

```
% cc -o all main.o more.o rest.o -lm -lexc
```

The `-lm` option specifies the math library; the `-lexc` option specifies the exception library.

Compile and link modules with a single command when you want to optimize your program. Most compilers support increasing levels of optimization with the use of certain options. For example:

- The `-O0` option requests no optimization (usually for debugging purposes).
- The `-O1` option requests certain local (module-specific) optimizations.
- Cross-module optimizations must be requested with the `-ifo` option. Specifying `-O3` in addition to `-ifo` improves the quality of cross-module optimization. In this case, compiling multiple files in one operation allows the compiler to perform the maximum possible optimizations. The `-ifo` option produces one `.o` file for multiple source files.

2.5.2 Linking with the ld Command

Normally, you do not need to run the linker directly, but use the cc command to indirectly invoke the linker. Executables that need to be built solely from assembler objects can be built with the ld command.

The linker (ld) combines one or more object files (in the order specified) into one executable program file, performing relocation, external symbol resolutions, and all other processing required to make object files ready for execution. Unless you specify otherwise, the linker names the executable program file a.out. You can execute the program file or use it as input for another linker operation.

The as assembler does not automatically invoke the linker. To link a program written in assembly language, do either of the following:

- Assemble and link with one of the other compiler commands. The .s suffix of the assembly language source file automatically causes the compiler command to invoke the assembler.
- Assemble with the as command and then link the resulting object file with the ld command.

For information about the options and libraries that affect the linking process, see ld(1).

2.5.3 Specifying Libraries

When you compile your program on the Tru64 UNIX system, it is automatically linked with the C library, libc.so. If you call routines that are not in libc.so or one of the archive libraries associated with your compiler command, you must explicitly link your program with the library. Otherwise, your program will not be linked correctly.

You need to explicitly specify libraries in the following situations:

- **When compiling multilanguage programs**

  If you compile multilanguage programs, be sure to explicitly request any required run-time libraries to handle unresolved references. Link the libraries by specifying -l*string*, where *string* is an abbreviation of the library name.

  For example, if you write a main program in C and some procedures in another language, you must explicitly specify the library for that language and the math library. When you use these options, the linker replaces the -l with lib and appends the specified characters (for the language library and for the math library) and the .a or .so suffix, depending upon whether it is a static (nonshared archive library) or dynamic (call-shared object or shared library) library. Then, it searches the following directories for the resulting library name:

```
/usr/shlib
/usr/ccs/lib
/usr/lib/cmplrs/cc
/usr/lib
/usr/local/lib
/var/shlib
```

For a list of the libraries that each language uses, see the reference pages of the compiler drivers for the various languages.

- **When storing object files in an archive library**

  You must include the pathname of the library on the compiler or linker command line. For example, the following command specifies that the `libfft.a` archive library in the `/usr/jones` directory is to be linked along with the math library:

  ```
  % cc main.o more.o rest.o /usr/jones/libfft.a -lm
  ```

  The linker searches libraries in the order that you specify. Therefore, if any file in your archive library uses data or procedures from the math library, you must specify the archive library before you specify the math library.

2.5.4 Avoiding Problems with Link Errors in Linker Output Files

For many types of link errors, the linker will create an output file in spite of the errors, overwriting any pre-existing output file with the same name. For most applications, this behavior does not create any problems. However, in those instances where it may cause problems, you can avoid overwriting the pre-existing output file by sending linker output to a temporary file and then — when the link is completed without errors — moving the temporary file to the desired output file. The following example shows how this is done for a call_shared executable file:

```
cc -o main.exe.X main.o
mv main.exe.X main.exe
```

The same technique can be applied when building shared libraries, as shown in the following example. In this case, you must use the `-soname` option to set the library’s internal name correctly.

```
ld -shared -soname libfoo.so -o libfoo.so.X foo.o -lc
mv libfoo.so.X libfoo.so
```

2.6 Running Programs

To run an executable program in your current working directory, in most cases you enter its file name. For example, to run the program `a.out` located in your current directory, enter:

```
% a.out
```

If the executable program is not in a directory in your path, enter the directory path before the file name, or enter:

```
% ./a.out
```

When the program is invoked, the `main` function in a C program can accept arguments from the command line if the `main` function is defined with one or more of the following optional parameters:

```
int main ( int argc, char *argv[ ], char *envp[ ] )[...]
```

The *argc* parameter is the number of arguments in the command line that invoked the program. The *argv* parameter is an array of character strings containing the arguments. The *envp* parameter is the environment array containing process information, such as the user name and controlling terminal. (The *envp* parameter has no bearing on passing command-line arguments. Its primary use is during `exec` and `getenv` function calls.)

You can access only the parameters that you define. For example, the following program defines the *argc* and *argv* parameters to echo the values of parameters passed to the program:

```
/*
 * Filename: echo-args.c
 * This program echoes command-line arguments.
*/

#include <stdio.h>

int main( int argc, char *argv[] )
{
int i;

printf( "program: %s\n", argv[0] ); /* argv[0] is program name */

for ( i=1; i < argc; i++ )
 printf( "argument %d: %s\n", i, argv[i] );

return(0);
}
```

The program is compiled with the following command to produce a program file called `a.out`:

```
$ cc echo-args.c
```

When the user invokes a.out and passes command-line arguments, the program echoes those arguments on the terminal. For example:

```
$ a.out Long Day\’s "Journey into Night"
     program: a.out
     argument 1: Long
     argument 2: Day’s
     argument 3: Journey into Night
```

The shell parses all arguments before passing them to a.out. For this reason, a single quote must be preceded by a backslash, alphabetic arguments are delimited by spaces or tabs, and arguments with embedded spaces or tabs are enclosed in quotation marks.

2.7 Object File Tools

After a source file has been compiled, you can examine the object file or executable file with following tools:

- odump — Displays the contents of an object file, including the symbol table and header information.
- stdump — Displays symbol table information from an object file.
- nm — Displays only symbol table information.
- file — Provides descriptive information on the general properties of the specified file, for example, the programming language used.
- size — Displays the size of the text, data, and bss segments.
- dis — Disassembles object files into machine instructions.

The following sections describe these tools. In addition, see strings(1) for information on using the strings command to find the printable strings in an object file or other binary file.

2.7.1 Dumping Selected Parts of Files (odump)

The odump tool displays header tables and other selected parts of an object or archive file. For example, odump displays the following information about the file echo-args.o:

```
% odump -at echo-args.o


   ***ARCHIVE SYMBOL TABLE***




   ***ARCHIVE HEADER***
 Member Name        Date       Uid     Gid     Mode      Size
```

```
   ***SYMBOL TABLE INFORMATION***
[Index] Name Value Sclass Symtype Ref
echo-args.o:
[0]  main 0x0000000000000000 0x01 0x06 0xfffff
[1]  printf 0x0000000000000000 0x06 0x06 0xfffff
[2]  _fpdata 0x0000000000000000 0x06 0x01 0xfffff
```

For more information, see odump(1).

2.7.2 Listing Symbol Table Information (nm)

The nm tool displays symbol table information for object files. For example, nm displays the following information about the object file produced for the executable file a.out:

```
% nm
nm: Warning: - using a.out

Name                            Value        Type       Size

.bss                   | 0000005368709568 | B | 0000000000000000
.data                  | 0000005368709120 | D | 0000000000000000
.lit4                  | 0000005368709296 | G | 0000000000000000
.lit8                  | 0000005368709296 | G | 0000000000000000
.rconst                | 0000004831842144 | Q | 0000000000000000
.rdata                 | 0000005368709184 | R | 0000000000000000
.
.
.
```

The Name column contains the symbol or external name; the Value column shows the address of the symbol, or debugging information; the Type column contains a letter showing the symbol type; and the Size column shows the symbol's size (accurate only when the source file is compiled with the debugging option, for example, −g). Some of the symbol type letters are:

- B — External zeroed data
- D — External initialized data
- G — External small initialized data
- Q — Read-only constants
- R — External read-only data

For more information, see nm(1).

2.7.3 Determining a File's Type (file)

The file command reads input files, tests each file to classify it by type, and writes the file's type to standard output. The file command uses the

/etc/magic file to identify files that contain a magic number. (A magic number is a numeric or string constant that indicates a file's type.)

The following example shows the output of the file command on a directory containing a C source file, object file, and executable file:

```
% file *.*
.:       directory
..:      directory
a.out:   COFF format alpha dynamically linked, demand paged executable
or object module not stripped - version 3.11-8
echo-args.c:    c program text
echo-args.o:    COFF format alpha executable or object module not
stripped - version 3.12-6
```

For more information, see file(1).

2.7.4 Determining a File's Segment Sizes (size)

The size tool displays information about the text, data, and bss segments of the specified object or archive file or files in octal, hexadecimal, or decimal format. For example, when it is called without any arguments, the size command returns information on a.out. You can also specify the name of an object or executable file on the command line. For example:

```
% size
text    data    bss    dec      hex
8192    8192    0      16384    4000
% size echo-args.o
text    data    bss    dec      hex
176     96      0      272      110
```

For more information, see size(1).

2.7.5 Disassembling an Object File (dis)

The dis tool disassembles object file modules into machine language. For example, the dis command produces the following output when it disassembles the a.out program:

```
% dis a.out
⋮

       __start:
 0x120001080:  23defff0        lda      sp, -16(sp)
 0x120001084:  b7fe0008        stq      zero, 8(sp)
 0x120001088:  c0200000        br       t0, 0x12000108c
 0x12000108c:  a21e0010        ldl      a0, 16(sp)
 0x120001090:  223e0018        lda      a1, 24(sp)
⋮
```

For more information, see dis(1).

2.8 ANSI Name Space Pollution Cleanup in the Standard C Library

The ANSI C standard states that users whose programs link against libc are guaranteed a certain range of global identifiers that can be used in their programs without danger of conflict with, or preemption of, any global identifiers in libc.

The ANSI C standard also reserves a range of global identifiers that libc can use in its internal implementation. These are called reserved identifiers and consist of the following, as defined in ANSI document number X3.159-1989:

- Any external identifier beginning with an underscore
- Any external identifier beginning with an underscore followed by an uppercase letter or an underscore

ANSI conformant programs are not permitted to define global identifiers that either match the names of ANSI routines or fall into the reserved name space specified earlier in this section. All other global identifier names are available for use in user programs.

Historical libc implementations contain large numbers of non-ANSI, nonreserved global identifiers that are both documented and supported. These routines are often called from within libc by other libc routines, both ANSI and otherwise. A user's program that defines its own version of one of these non-ANSI, nonreserved items would pre-empt the routine of the same name in libc. This could alter the behavior of supported libc routines, both ANSI and otherwise, even though the user's program may be ANSI-conformant. This potential conflict is known as ANSI name space pollution.

The implementation of libc on Tru64 UNIX includes a large number of non-ANSI, nonreserved global identifiers that are both documented and supported. To protect against preemption of these global identifiers within libc and to avoid pollution of the user's name space, the vast majority of these identifiers have been renamed to the reserved name space by prepending two underscores (_ _) to the identifier names. To preserve external access to these items, weak identifiers have been added using the original identifier names that correspond to their renamed reserved counterparts. Weak identifiers work much like symbolic links between files. When the weak identifier is referenced, the strong counterpart is used instead.

User programs linked statically against libc may have extra symbol table entries for weak identifiers. Each of these identifiers will have the same address as its reserved counterpart, which will also be included in the symbol table. For example, if a statically linked program simply called the tzset() function from libc, the symbol table would contain two entries for this call, as follows:

```
# stdump -b a.out | grep tzset
18. (file 9) (4831850384) tzset Proc Text symref 23 (weakext)
39. (file 9) (4831850384) __tzset Proc Text symref 23
```

In this example, tzset is the weak identifier and __tzset is its strong counterpart. The __tzset identifier is the routine that will actually do the work.

User programs linked as shared should not see such additions to the symbol table because the weak/strong identifier pairs remain in the shared library.

Existing user programs that reference non-ANSI, nonreserved identifiers from libc do not need to be recompiled because of these changes, with one exception: user programs that depended on preemption of these identifiers in libc will no longer be able to pre-empt them using the nonreserved names. This kind of preemption is not ANSI-compliant and is highly discouraged. However, the ability to pre-empt these identifiers still exists by using the new reserved names (those preceded by two underscores).

These changes apply to the dynamic and static versions of libc:

- /usr/shlib/libc.so
- /usr/lib/libc.a

When debugging programs linked against libc, references to weak symbols resolve to their strong counterparts, as in the following example:

```
% dbx a.out
dbx version 3.11.4

Type 'help' for help.

main:   4   tzset

(dbx) stop in tzset

[2] stop in __tzset

(dbx)
```

When the weak symbol tzset in libc is referenced, the debugger responds with the strong counterpart __tzset instead because the strong counterpart actually does the work. The behavior of the dbx debugger is the same as if __tzset were referenced directly.

2.9 Inline Assembly Code — ASMs

The compiler supports in-line assembly code, commonly called ASMs.

Like built-in functions, ASMs are implemented with a function-call syntax. But unlike built-in functions, to use ASMs you must include the <c_asm.h> header file containing prototypes for the three types of ASMs, and the #pragma intrinsic preprocessor directive.

These functions have the following format:

```
__int64 asm(const char *, ...); /* for integer operations, like MULQ */
float fasm(const char *, ...); /* for single precision float instructions, like MULS */
double dasm(const char *, ...); /* for double precision float instructions, like MULT */
#pragma intrinsic (asm, fasm, dasm)
```

const char*

The first argument to the asm, fasm, or dasm function contains the instructions to be generated inline and the metalanguage that describes the interpretation of the arguments.

...

The source and destination arguments (if any) for the instruction being generated, and any other values used in the generated instructions.

These values are made available to the instructions through the normal argument passing conventions of the calling standard (the first integer argument is available in register R16).

The #pragma intrinsic directive in the <c_asm.h> header file is required when using ASMs. It notifies the compiler that:

- These functions are not user-defined functions.
- The special ASM processing should be applied to analyze at compile time the first argument and generate machine-code instructions as specified by the contents of the string.

The metalanguage for the argument references has the following form:

```
<metalanguage_sequence> : <register_alias>
                         | <register_number>
                         | <register_macro>
                         ;

 <register_number>       : "$" number
                         ;

 <register_macro>        : "%" <macro_sequence>
                         ;
 <macro_sequence>        : number
                         | <register_name>
                         | "f" number | "F" number
                         | "r" number | "R" number
                         ;

 <register_name> :     /* argument registers: R16-R21 */
                      "a0" | "a1" | "a2" | "a3" | "a4" | "a5"

                      /* return value: R0 or F0, depending on type */
                    | "v0"

                      /* scratch registers: R1, R22-R24, R28 */
                    | "t0" | "t1" | "t2" | "t3" | "t4"

                      /* save registers: R2-R15 */
                    | "s0" | "s1" | "s2" | "s3" | "s4"  | "s5"  | "s6"  | "s7"
                    | "s8" | "s7" | "s8" | "s9" | "s10" | "s11" | "s12" | "s13"

                      /* stack pointer: R30 */
                    | "sp" | "SP" | "$sp" | "$SP"

                    | "RA" | "ra"           /* return addr:        R26  */
                    | "PV" | "pv"           /* procedure value:    R27  */
                    | "AI" | "ai"           /* arg info:           R25  */
                    | "FP" | "fp"           /* frame pointer:      R29  */
                    | "RZ" | "rz" | "zero"  /* sink/source: R31 == zero */
```

Syntactically, the metalanguage can appear anywhere within an instruction sequence.

The literal string that contains instructions, operands, and metalanguage must follow the general form:

```
<string_contents>       :  <instruction_seq>
                         |  <string_contents> ";" <instruction_seq>
                         |  error
                         |  <string_contents> error
                         ;

 <instruction_seq>       :  instruction_operand
                         |  directive
                         ;
```

An instruction_operand is generally recognized as an assembly language instruction separated by whitespace from a sequence of comma-separated operands.

Because the C language concatenates adjacent string literals into a single string, successive instructions can be written as separate strings, one per line (as is normally done in assembly language) as long as each instruction is terminated by a semicolon (as shown in the examples).

There are semantic and syntax rules associated with ASMs:

- The first argument to an ASM call is interpreted as the instructions to be assembled in the metalanguage, and must be fully understood by the compiler at compile time. Therefore, it must be a literal string (or a macro expanding to a literal string) and must not be a run-time value containing a string. Therefore, the following are not allowed: indirections, table lookups, structure dereferences, and so on.

- The remaining arguments are loaded into the argument registers like normal function arguments, except that the second argument to the ASM call is treated as the first argument for purposes of the calling standard.

For example, in the following test, the six arguments are loaded into arg registers `a0` through `a5`, and the result of each subexpression is stored in the value return register `v0`. Since `v0` is the calling standard's return value register (`R0` for an integer function), the result of the final `MULQ` is the value returned by the "call":

```
if (asm("mulq %a0, %a1,  %v0;"
            "mulq %a2, %v0,  %v0;"
            "mulq %a3, %v0,  %v0;"
            "mulq %a4, %v0,  %v0;"
            "mulq %a5, %v0,  %v0;", 1, 2, 3, 4, 5, 6) != 720){
      error_cnt++;
      printf ("Test failed\n");
         }
```

The following example does not work. There is no value loaded into the floating-point return register. Furthermore, it results in a compile-time warning stating that r2 is used before it is set, because the arguments are loaded into the arg registers and not into r2:

```
z =  fasm("mulq %r2, %a1 %r5", x=10, y=5);
```

The correct way of doing this is to specify an argument register number in place of r2. A correct version of the above would be:

```
z = fasm("mulq   %a0, %a1, %a1;"
          "stq    %a1, 0(%a2);"
          "ldt    %f0, 0(%a2);"
          "cvtqf  %f0, %f0;",  x=10, y=5, &temp);
```

Note that the memory location used for the transfer from integer to floating-point register is made available to the `asm` code by passing as an argument the address of a variable allocated in the C code for that purpose.

- A return register must be specified in the metalanguage for the result to appear in the expected place.
- For intructions that do not take any argument and do not have a return type, leave out the arguments. For example:

```
asm("MB");
```

3 Pragma Preprocessor Directives

The #pragma directive is a standard method for implementing features that vary from one compiler to the next. This chapter describes the implementation-specific pragmas that are supported on the C compiler:

- #pragma assert (Section 3.1)
- #pragma environment (Section 3.2)
- #pragma extern_model (Section 3.3)
- #pragma extern_prefix (Section 3.4)
- #pragma inline (Section 3.5)
- #pragma intrinsic and #pragma function (Section 3.6)
- #pragma linkage (Section 3.7)
- #pragma member_alignment (Section 3.8)
- #pragma message (Section 3.9)
- #pragma optimize (Section 3.10)
- #pragma pack (Section 3.11)
- #pragma pointer_size (Section 3.12)
- #pragma unroll (Section 3.13)
- #pragma use_linkage (Section 3.14)
- #pragma weak (Section 3.15)

Pragmas supported by all implementations of Compaq C are described in the *Compaq C Language Reference Manual*.

Some pragmas may perform macro expansion, by default. The *Compaq C Language Reference Manual* lists these pragmas. It also describes the use of double underscores as prefixes to pragma names and keywords to avoid macro expansion problems when porting a program that defines a macro with the same name as a pragma name or keyword.

You can force macro expansion for any pragma by adding a _m suffix to the pragma name. When the pragma name is followed by _m, the text that follows the pragma name is subject to macro replacement. See the example in Section 3.1.3.

3.1 The #pragma assert Directive

The `#pragma assert` directive allows you to specify assertions about a program that the compiler can then use to generate more efficient code.

The `#pragma assert` directive is never needed to make a program execute correctly; however, if a `#pragma assert` is specified, the assertions must be valid or the program might behave incorrectly.

The `#pragma assert` directive has the following formats:

#pragma assert func_attrs(*identifier-list***)***function-assertions*

#pragma assert global_status_variable(*variable-list***)**

#pragma assert non_zero(*constant-expression***) "***string-literal***"**

3.1.1 #pragma assert func_attrs

Use this form of the `#pragma assert` directive to make assertions about a function's attributes.

This form of the pragma has the following format:

#pragma assert func_attrs(*identifier-list***)***function-assertions*

identifier-list

A list of function identifiers about which the compiler can make assumptions based on the *function-assertions*. If you specify more than one identifier, separate them with commas.

function-assertions

A list of assertions that the compiler uses to make assumptions about the functions. Specify one or more of the following assertions, separating multiple assertions with white space:

`noreturn`

The compiler can assume that any call to the routine will never return.

`nocalls_back`

The compiler can assume that no routine in the source module will be called before control is returned from this function.

`nostate`

The compiler can assume that only the function's arguments determine the value that the function returns and any side-effects the function might have. If a function is marked as having both noeffects and nostate, the compiler can eliminate redundant calls to the function.

`noeffects`

The compiler can assume that any call to this function will have no effect except to set the return value of the function. If the compiler determines that the return value from a function call is never used, it can remove the call.

`file_scope_vars(`*`option`*`)`

The compiler can make assumptions about how a function will access variables that are declared at file scope (with either internal or external linkage). The *`option`* is one of the following keywords:

`none`

The function will not read or write to any file-scope variables except those whose type is `volatile` or those that are listed in a `#pragma assert global_status_variable`.

`noreads`

The function will not read any file-scope variables except those whose type is `volatile` or those that are listed in a `#pragma assert global_status_variable`.

`nowrites`

The function will not write to any file-scope variables except those whose type is `volatile` or those that are listed in a `#pragma assert global_status_variable`.

`format (`*`style`*`, `*`format-index`*`, `*`first-to-check-index`*`)`

The compiler can assume that this function takes `printf` or `scanf` style arguments which should be type-checked against a format string.

The format attribute allows you to identify your own functions which take format strings as arguments, so that the compiler can check the calls to these functions for errors. The compiler checks formats for the library functions `printf`, `fprintf`, `sprintf`, `snprintf`, `scanf`, `fscanf`, and `sscanf` whenever these functions are enabled as intrinsics (the default). You may use the format attribute to assert that the formats of these functions should be checked when they are not enabled as intrinsics.

`style`

Determines how the format string is interpreted, and should be either `printf` or `scanf`.

format-index

Specifies which argument is the format string argument (starting from 1).

first-to-check-index

The number of the first argument to check against the format string.

For functions where the arguments are not available to be checked (such as `vprintf`), specify the third parameter as zero. In this case the compiler only checks the format string for consistency. For example, the following declaration causes the compiler to check the arguments in calls to `your_printf` for consistency with the `printf` style format string argument `your_format`:

```
extern int
your_printf (void *your_object, const char *your_format, ...);
#pragma assert func_attrs(your_printf) format (printf, 2, 3)
```

The format string (`your_format`) is the second argument of the function `your_printf`, and the arguments to check start with the third argument, so the correct parameters for the format attribute are 2 and 3.

This form of the `#pragma assert` directive must appear at file scope.

The identifiers in the *`identifier-list`* must have declarations that are visible at the point of the `#pragma assert` directive.

A function can appear on more than one `#pragma assert func_attrs` directive as long as each directive specifies a different assertion about the function. For example, the following is valid:

```
#pragma assert func_attrs(a) nocalls_back
#pragma assert func_attrs(a) file_scope_vars(noreads)
```

But the following is not valid:

```
#pragma assert func_attrs(a) file_scope_vars(noreads)
#pragma assert func_attrs(a) file_scope_vars(nowrites)
```

3.1.2 #pragma assert global_status_variable

Use this form of the `#pragma assert` directive to specify variables that are to be considered global status variables, which are exempt from any assertions that `#pragma assert func_attrs file_scope_vars` directives give to functions.

This form of the pragma has the following syntax:

#pragma assert global_status_variable(*variable-list***)**

The *variable-list* is a list of variables.

This form of the #pragma assert directive must appear at file scope.

The variables in the *variable-list* must have declarations that are visible at the point of the #pragma assert directive.

3.1.3 #pragma assert non_zero

This form of the #pragma assert directive has the following syntax:

#pragma assert non_zero(*constant-expression***) "***string-literal***"**

When the compiler encounters this directive, it evaluates the *constant-expression*. If the expression is zero, the compiler generates a message that contains both the specified string-literal and the compile-time constant. For example:

```
#pragma assert non_zero(sizeof(a) == 12) "a is the wrong size"
```

If the compiler determines that the sizeof a is not 12, it generates the following message:

```
cc: Warning: a.c, line 4: The assertion "sizeof(a)==12" was
    not true, a is the wrong size. (assertfail)
```

Unlike the assert options func_attrs and global_status_variable, #pragma assert non_zero can appear either inside or outside a function body. When it is used inside a function body, the #pragma can appear where a statement can appear, but it is not treated as a statement. When it is used outside a function body, the #pragma can appear where a declaration can appear, but it is not treated as a declaration.

Any variables in the *constant-expression* must have declarations that are visible at the point of the #pragma assert directive.

Because #pragma assert does not perform macro replacement on the pragma, it is often necessary to use the #pragma assert_m directive. Consider the following program that verifies both the size of a struct and the offset of one of its elements.

```
#include <stddef.h>
typedef struct {
   int a;
   int b;
} s;
#pragma assert non_zero(sizeof(s) == 8) "sizeof assert failed"
#pragma assert_m non_zero(offsetof(s,b) == 4) "offsetof assert failed"
```

Because offsetof is a macro, the second pragma must be assert_m so that offsetof will expand correctly.

3.2 The #pragma environment Directive

The #pragma environment directive allows you to set, save, and restore the state of all context pragmas. The context pragmas are:

```
#pragma extern_model
#pragma extern_prefix
#pragma member_alignment
#pragma message
#pragma pack
#pragma pointer_size
```

A context pragma can save and restore previous states, usually before and after including a header file that might also use the same type of pragma.

The #pragma environment directive protects include files from compilation contexts set by encompassing programs, and protects encompassing programs from contexts set in header files that they include.

This pragma has the following syntax:

#pragma environment [command_line | header_defaults | restore | save]

command_line

Sets the states of all of the context pragmas set on the command line. You can use this pragma to protect header files from environment pragmas that take effect before the header file is included.

header_defaults

Sets the states of all of the context pragmas to their default values. This is equivalent to the situation in which a program with no command-line options and no pragmas is compiled, except that this pragma sets the pragma message state to #pragma nostandard, as is appropriate for header files.

restore

Restores the current state of every context pragma.

save

Saves the current state of every context pragma.

Without requiring further changes to the source code, you can use #pragma environment to protect header files from things such as language enhancements that might introduce additional compilation contexts.

A header file can selectively inherit the state of a pragma from the including file and then use additional pragmas as needed to set the compilation to nondefault states. For example:

```
#ifdef __pragma_environment
#pragma __environment save 1
#pragma __environment header_defaults 2
#pragma member_alignment restore 3
#pragma member_alignment save 4
#endif
.
.    /*contents of header file*/
.
#ifdef __pragma_environment
#pragma __environment restore
#endif
```

In this example:

1. Saves the state of all context pragmas.
2. Sets the default compilation environment.
3. Pops the member alignment context from the `#pragma member_alignment` stack that was pushed by `#pragma __environment save`, restoring the member alignment context to its pre-existing state.
4. Pushes the member alignment context back onto the stack so that the `#pragma __environment restore` can pop the entry.

Therefore, the header file is protected from all pragmas, except for the member alignment context that the header file was meant to inherit.

3.3 The #pragma extern_model Directive

The `#pragma extern_model` directive controls how the compiler interprets data objects that have external linkage. With this pragma you can select one of the following global symbol models to be used for external (`extern`) objects:

`relaxed_refdef`

> In this model, some declarations are references and some are definitions. Multiple uninitialized definitions for the same object are allowed, and the linker resolves them into one. However, a reference requires that at least one definition exists. C compilers on most UNIX systems use this model; it is the default model on Compaq C.

`strict_refdef`

In this model, some declarations are references and some are definitions. There must be exactly one definition in the program for any symbol that is referenced. This model is the only one guaranteed to strictly conform to all ANSI C implementations.

Note

Compaq C on HP OpenVMS platforms supports two other external models named `common_block` and `globalvalue`, but these are not supported on Tru64 UNIX.

After a global symbol model is selected with the `extern_model` pragma, all subsequent declarations of objects that have external storage class are treated according to the specified model until another `extern_model` pragma is specified.

For example, consider the following pragma:

```
#pragma extern_model strict_refdef "progsec3"
```

After this pragma is specified, the following file-level declarations are treated as declaring global symbols according to the `strict_refdef` model:

```
int x = 0;
extern int y;
```

Regardless of the external model, the compiler uses ANSI C rules to determine if a declaration is a definition or a reference. An external definition is a file-level declaration that has no storage-class keyword, or that contains the `extern` storage-class keyword, and is also initialized. A reference is a declaration that uses the `extern` storage-class keyword and is not initialized. In the previous example, the declaration of `x` is a global definition and the declaration of `y` is a global reference.

A stack of the compiler's external model state is kept so that `#pragma extern_model` can be used transparently in header files and in small regions of program text. See Section 3.3.4 and Section 3.3.5 for more information.

The following sections describe the various forms of the `#pragma extern_model` directive.

3.3.1 Syntax

The `#pragma extern_model` directive has the following syntax:

#pragma extern_model *model_spec* **[***attr***[,***attr***]...]**

model_spec

One of the following:

```
relaxed_refdef
strict_refdef "name"
```

where "*name*" is the name of the program section (psect) for any definitions.

[*attr*[,*attr*]...]

Optional psect attribute specifications. Choose only one from each of the following sets of attribute specifications:

shr|noshr

The psect can be shared in memory (shr) or cannot be shared in memory (noshr). The default is noshr.

wrt|nowrt

The psect contains data that can be modified (wrt) or data that cannot be modified (nowrt). The default is determined by the first variable placed in the psect. If the variable has the const type qualifier (or the readonly modifier), the psect is set to nowrt. Otherwise, it is set to wrt.

ovr|con

The psect is concatenated with other psects with the same name (con) or overlaid on the same memory locations (ovr). The default is con for strict_refdef, and over for relaxed_refdef.

4|octa|5|6|7|8|9|10|11|12|13|14|15|16|page

These denote numeric alignment values. The default alignment is octa. If a number is specified, the psect is given an alignment of two raised to the power indicated by that number.

The strict_refdef extern_model can also take the following psect attribute specification:

noreorder

Causes variables in the section to be allocated in the order they are defined. This attribute specification is off, by default.

In the following example, the initialized variables are to be aligned on a 64K-byte (2**16) boundary. The `noreorder` attribute specification means that the variables will be allocated in the order they are declared:

```
#pragma extern_model save
#pragma extern_model strict_refdef "progsecA" 16,noreorder
int var1 = 5;
int var2 = 6;
#pragma extern_model restore
```

In the following example, the (non-writable) variables are to be aligned on a data cache line boundary:

```
#pragma extern_model save
#pragma extern_model strict_refdef "progsecB" 3,noreorder,nowrt
const long c_v1 = 1;
const long c_v2 = 2;
const long c_v3 = 2;
const long c_v4 = 2;
#pragma extern_model restore
```

Under the `relaxed_refdef` model, psect attributes do not affect variables that are declared with tentative definitions. Consider the following code:

```
#pragma extern_model relaxed_refdef 5
int a;
int b=6;
#pragma extern_model strict_refdef 5
int c;
```

Variable `a` is given the default octaword (2**4 or 16-byte) alignment because it is a tentative definition. But `b` is given 32-byte (2**5) alignment because it is initialized. Although `c` is a tentative definition, it is 32-byte (2**5) aligned because it is under the `strict_refdef` model.

Note

The psect attributes are normally used by system programmers who need to perform declarations normally done in macro. Most of these attributes are not needed in normal C programs.

3.3.2 #pragma extern_model relaxed_refdef

This pragma sets the compiler's model of external data to the `relaxed_refdef` model, which is the one used on UNIX systems.

The `#pragma extern_model relaxed_refdef` directive has the following syntax:

```
#pragma extern_model relaxed_refdef [attr[,attr]...]
```

3.3.3 #pragma extern_model strict_refdef

This pragma sets the compiler's model of external data to the `strict_refdef` model. Use this model for a program that is to be an ANSI C strictly conforming program.

The `#pragma extern_model strict_refdef` directive has the following syntax:

```
#pragma extern_model strict_refdef "name" [attr[,attr]...]
```

The *name* in quotes, if specified, is the name of the psect for any definitions.

3.3.4 #pragma extern_model save

This pragma pushes the current external model of the compiler onto a stack. The stack records all information that is associated with the external model, including the `shr/noshr` state and any quoted psect name.

This pragma has the following syntax:

```
#pragma extern_model save
```

The number of entries that are allowed in the `#pragma extern_model` stack is limited only by the amount of memory that is available to the compiler.

3.3.5 #pragma extern_model restore

This pragma pops the external model stack of the compiler. The external model is set to the state popped off the stack. The stack records all information that is associated with the external model, including the `shr/noshr` state and any quoted psect name.

This pragma has the following syntax:

```
#pragma extern_model restore
```

On an attempt to pop an empty stack, a warning message is issued and the compiler's external model is not changed.

3.4 The #pragma extern_prefix Directive

The `#pragma extern_prefix` directive controls the compiler's synthesis of external names, which the linker uses to resolve external name requests.

When you specify `#pragma extern_prefix` with a string argument, the C compiler attaches the string to the beginning of all external names produced by the declarations that follow the pragma specification.

This pragma is useful for creating libraries where the facility code can be attached to the external names in the library.

This pragma has the following syntax:

#pragma extern_prefix "*string*" [(*id*,...)]

#pragma extern_prefix save

#pragma extern_prefix restore

The quoted *string* is attached to external names in the declarations that follow the pragma specification.

You can also specify a prefix for specific external identifiers using the optional list [(*id*,...)].

You can use the `save` and `restore` keywords can be used to save the current pragma prefix string and to restore the previously saved pragma prefix string, respectively.

The default prefix for external identifiers, when none has been specified by a pragma, is the null string.

The recommended use is as follows:

```
#pragma extern_prefix save
#pragma extern_prefix " prefix-to-prepend-to-external-names "
...some declarations and definitions ...
#pragma extern_prefix restore
```

When an `extern_prefix` is in effect and you are using `#include` to include header files, but do not want the `extern_prefix` to apply to `extern` declarations in the header files, use the following code sequence:

```
#pragma extern_prefix save
#pragma extern_prefix ""
#include ...
#pragma extern_prefix restore
```

Otherwise, the prefix is attached to the beginning of external identifiers for definitions in the included files.

Notes

The following notes apply when specifying optional identifiers on `#pragma extern_prefix`:

- For each *id* there must not be a declaration of that *id* visible at the point of the pragma; otherwise a warning is issued, and there is no effect on that *id*.
- Each *id* affected by a pragma with a nonempty prefix is expected to be subsequently declared with external linkage

in the same compilation unit. The compiler issues a default informational if there is no such declaration made by the end of the compilation.

- It is perfectly acceptable for the *id* list form of the pragma, or declarations of the *id*s listed, to occur within a region of source code controlled by the other form of the pragma. The two forms do not interact; the form with an *id* list always supersedes the other form.
- There is no interaction between the `save`/`restore` stack and the *id* lists.
- If the same *id* appears in more than one pragma, then a default informational message is issued, unless the prefix on the second pragma is either empty (`""`) or matches the prefix from the previous pragma. In any case, the behavior is that the last-encountered prefix supersedes all others.

3.5 The #pragma inline Directive

Function inlining is the inline expansion of function calls, replacing the function call with the function code itself. Inline expansion of functions reduces execution time by eliminating function-call overhead and allowing the compiler's general optimization methods to apply across the expanded code. Compared with the use of function-like macros, function inlining has the following advantages:

- Arguments are evaluated only once.
- Overuse of parentheses is not necessary to avoid problems with precedence.
- Actual expansion can be controlled from the command line.
- The semantics are as if inline expansion had not occurred. You cannot get this behavior using macros.

The following preprocessor directives control function inlining:

#pragma inline (*id, ...***)**

#pragma noinline (*id, ...***)**

Where *id* is a function identifier:

- If a function is named in a `#pragma inline` directive, calls to that function are expanded as inline code, if possible.
- If a function is named in a `#pragma noinline` directive, calls to that function are not expanded as inline code.

- If a function is named in both a #pragma inline and a #pragma noinline directive, an error message is issued.

If a function is to be expanded inline, you must place the function definition in the same module as the function call (unless you also use the -ifo option, which allows inlining across modules). The definition can appear either before or after the function call.

The cc command options -O3, -O4, -inline size, -inline speed, or -inline all cause the compiler to attempt to expand calls to functions named in neither a #pragma inline nor a #pragma noinline directive as inline code whenever appropriate, as determined by the following function characteristics:

- Size
- Number of times the function is called
- Conformance to the following restrictions:
  - The function does not take a parameter's address.
  - A field of a struct argument. An argument that is a pointer to a struct is not restricted.
  - The function does not use the varargs or stdarg package to access the function's arguments because they require arguments to be in adjacent memory locations, and inline expansion may violate that requirement.

For optimization level -O2, the C compiler inlines small static routines only.

The #pragma inline directive causes inline expansion regardless of the size or number of times the specified functions are called.

3.6 The #pragma intrinsic and #pragma function Directives

Certain functions can be declared to be intrinsic. Intrinsic functions are functions for which the C compiler generates optimized code in certain situations, possibly avoiding a function call.

Table 3–1 shows the functions that can be declared to be intrinsic.

Table 3–1: Intrinsic Functions

abs	fabs	labs
printf	fprintf	sprintf
strcpy	strlen	memcpy
memmove	memset	alloca
bcopy	bzero	

To control whether a function is treated as an intrinsic, use one of the following directives (where *func_name_list* is a comma-separated list of function names enclosed in parentheses):

#pragma intrinsic (*func_name_list***)**

#pragma function (*func_name_list***)**

#pragma function ()

The #pragma intrinsic directive enables intrinsic treatment of a function. When the #pragma intrinsic directive is turned on, the compiler understands how the functions work and is thus able to generate more efficient code. A declaration for the function must be in effect at the time the pragma is processed.

The #pragma function directive disables the intrinsic treatment of a function. A #pragma function directive with an empty *func_name_list* disables intrinsic processing for all functions.

Some standard library functions also have built-in counterparts in the compiler. A built-in is a synonym name for the function and is equivalent to declaring the function to be intrinsic. The following built-ins (and their built-in names) are provided:

Function	Synonym
abs	__builtin_abs
labs	__builtin_labs
fabs	__builtin_fabs
alloca	__builtin_alloca
strcpy	__builtin_strcpy

Several methods are available for using intrinsics and built-ins. The header files containing the declarations of the functions contain the #pragma intrinsic directive for the functions shown in Table 3–1. To enable the directive, you must define the preprocessor macro _INTRINSICS. For alloca, all that is necessary is to include alloca.h.

For example, to get the intrinsic version of abs, a program should either include stdlib.h and compile with -D_INTRINSICS or define _INTRINSICS with a #define directive before including stdlib.h.

To enable built-in processing, use the -D switch. For example, to enable the fabs built-in, the proc.c program is compiled with one of the following:

```
% cc -Dfabs=__builtin_fabs prog.c
% cc -Dabs=__builtin_abs prog.c
```

Optimization of the preceding functions varies according to the function and how it is used:

- The following functions are inlined:

  ```
  abs
  fabs
  labs
  alloca
  ```

  The function call overhead is removed.
- In certain instances, the `printf` and `fprintf` functions are converted to call `puts`, `putc`, `fputs`, or `fputc` (or their equivalents), depending on the format string and the number and types of arguments.
- In certain instances, the `sprintf` function is inlined or converted to a call to `strcpy`.
- The `strcpy` function is inlined if the source string (the second argument) is a string literal.

3.7 The #pragma linkage Directive

The `#pragma linkage` directive allows you to specify linkage types. A linkage type specifies how a function uses a set of registers. It allows you to specify the registers that a function uses. It also allows you to specify the characteristics of a function (for example, the registers in which it passes parameters or returns values) and the registers that it can modify. The `#pragma use_linkage` directive associates a previously defined linkage with a function (see Section 3.14).

The `#pragma linkage` directive affects both the call site and function compilation (if the function is written in C). If the function is written in assembler, you can use the linkage pragma to describe how the assembler uses registers.

The `#pragma linkage` directive has the following format:

`#pragma linkage` *`linkage-name`* **`= (`***`characteristics`***`)`**

`linkage-name`

Identifies the linkage type being defined. It has the form of a C identifier. Linkage types have their own name space, so their names will not conflict with other identifiers or keywords in the compilation unit.

characteristics

Specifies information about where parameters will be passed, where the results of the function are to be received, and what registers are modified by the function call.

You must specify a *register-list*. A *register-list* is a comma-separated list of register names, either r*n* or f*n*. A *register-list* can also contain parenthesized sublists. Use the *register-list* to describe arguments and function result types that are structures, where each member of the structure is passed in a single register. For example:

```
parameters(r0,(f0,f1))
```

The preceding example is a function with two parameters. The first parameter is passed in register `r0`. The second parameter is a structure type with two floating-point members, which are passed in registers `f0` and `f1`.

The following list of *characteristics* can be specified as a parenthesized list of comma-separated items. Note, these keywords can be supplied in any order.

- `parameters (`*register-list*`)`

  The `parameters` characteristic passes arguments to a routine in specific registers.

  Each item in the *register-list* describes one parameter that is passed to the routine.

  You can pass structure arguments by value, with the restriction that each member of the structure is passed in a separate parameter location. Doing so, however, may produce code that is slower because of the large number of registers used. The compiler does not diagnose this condition.

  Valid registers for the `parameters` option include integer registers `r0` through `r25` and floating-point registers `f0` through `f30`.

  Structure types require at least one register for each field. The compiler verifies that the number of registers required for a structure type is the same as the number provided in the pragma.

- `result (`*register-list*`)`

  The compiler needs to know which registers will be used to return the value from the function. Use the `result` characteristic to pass this information.

  If a function does not return a value (that is, the function has a return type of `void`), do not specify `result` as part of the linkage.

Valid registers for the `register` option include general-purpose registers `r0` through `r25` and floating-point registers `f0` through `f30`.

-
  ```
  preserved (register-list)
  nopreserve (register-list)
  notused (register-list)
  notneeded ((lp))
  ```

  The compiler needs to know which registers are used by the function and which are not, and of those used, whether they are preserved across the function call. To specify this information, use the `preserved`, `nopreserve`, `notused`, and `notneeded` options:

  - A `preserved` register contains the same value after a call to the function as it did before the call.
  - A `nopreserve` register does not necessarily contain the same value after a call to the function as it did before the call.
  - A `notused` register is not used in any way by the called function.
  - The `notneeded` characteristic indicates that certain items are not needed by the routines using this linkage. The `lp` keyword specifies that the Linkage Pointer register (`r27`) does not need to be set up when calling the specified functions. The linkage pointer is required when the called function accesses global or `static` data. You must determine whether it is valid to specify that the register is not needed.

  Valid registers for the `preserved`, `nopreserve`, and `notused` options include general-purpose registers `r0` through `r30` and floating-point registers `f0` through `f30`.

The `#pragma linkage` directive does not support structures containing nested substructures as parameters or function return types with special linkages. Functions that have a special linkage associated with them do not support parameters or return types that have a union type.

The following characteristics specify a *`simple-register-list`* containing two elements (registers `f3` and `f4`); and a *`register-list`* containing two elements (register `r0` and a sublist containing the registers `f0` and `f1`):

```
nopreserve(f3,f4)
parameters(r0,(f0,f1))
```

The following example shows a linkage using such characteristics:

```
#pragma linkage my_link=(nopreserve(f3,f4),
                parameters(r0,(f0,f1)),
                notneeded (lp))
```

The parenthesized notation in a *`register-list`* describes arguments and function return values of type `struct`, where each member of the `struct` is passed in a single register. In the following example, `sample_linkage` specifies two parameters — the first is passed in registers `r0`, `r1`, and `r2` and the second is passed in `f1`:

```
struct sample_struct_t {
    int A, B;
    short C;
    } sample_struct;

#pragma linkage sample_linkage = (parameters ((r0, r1, r2), f1))
void sub (struct sample_struct_t p1, double p2) { }

main()
{
    double d;

    sub (sample_struct, d);
}
```

3.8 The #pragma member_alignment Directive

By default, the compiler aligns structure members on natural boundaries. Use the `#pragma [no]member_alignment` preprocessor directive to determine the byte alignment of structure members.

This pragma has the following formats:

`#pragma member_alignment` [**`save | restore`**]

`#pragma nomember_alignment` [***`base_alignment`***]

`save | restore`

Saves the current state of the member alignment (including pack alignment) and restores the previous state, respectively. The ability to control the state is necessary for writing header files that require `member_alignment` or `nomember_alignment`, or that require inclusion in a `member_alignment` that is already set.

base_alignment

The *base_alignment* parameter can be used to specify the base-alignment of the structure. Use one of the following keywords for *base_alignment*:

- `byte` (1 byte)

- word (2 bytes)
- longword (4 bytes)
- quadword (8 bytes)
- octaword (16 bytes)

Use #pragma member_alignment to specify natural-boundary alignment of structure members. When #pragma member_alignment is used, the compiler aligns structure members on the next boundary appropriate to the type of the member, rather than on the next byte. For example, an int variable is aligned on the next longword boundary and a short variable is aligned on the next word boundary.

Use #pragma nomember_alignment to specify byte alignment of structure members.

To specify structure member alignment on byte, word, longword, or quadword boundaries, use the pragma pack directive. See Section 3.11 for more information on #pragma pack.

With any combination of #pragma member_alignment, #pragma nomember_alignment, and #pragma pack, each pragma remains in effect until the next one is encountered.

3.9 The #pragma message Directive

The #pragma message directive controls the issuance of individual diagnostic messages or groups of diagnostic messages. The use of this pragma overrides any command-line options that may affect the issuance of messages.

The #pragma message directive has the following formats:

#pragma message *option1* **(***message-list***)**

#pragma message *option2*

#pragma message ("*string***")**

3.9.1 #pragma message option1

This form of the #pragma message directive has the following format:

#pragma message *option1* **(***message-list***)**

The *option1* parameter must be one of the following keywords:

enable

Enables issuance of the messages specified in the message list.

`disable`

Disables issuance of the messages specified in the message list. Only messages of severity Warning or Information can be disabled. If the message has a severity of Error or Fatal, it is issued regardless of any attempt to disable it.

`emit_once`

Emits the specified messages only once per compilation. Certain messages are emitted only the first time the compiler encounters the causal condition. When the compiler encounters the same condition later in the program, no message is emitted. Messages about the use of language extensions are an example of this kind of message. To emit one of these messages every time the causal condition is encountered, use the `emit_always` option.

Errors and Fatals are always emitted. You cannot set them to `emit_once`.

`emit_always`

Emits the specified messages at every occurrence of the condition.

`error`

Sets the severity of the specified messages to Error. Supplied Error messages and Fatal messages cannot be made less severe. (The exception is that a message can be upgraded from Error to Fatal, then later downgraded to Error again, but it can never be downgraded from Error. Warnings and Informationals can be made any severity.)

`fatal`

Sets the severity of the specified messages to Fatal.

`informational`

Sets the severity of the specified messages to Informational. Note that Fatal and Error messages cannot be made less severe.

`warning`

Sets the severity of each message in the *`message-list`* to Warning. Note that Fatal and Error messages cannot be made less severe.

The *`message-list`* parameter can be one of the following:

Note

The default is to issue all diagnostic messages for the selected compiler mode except those in the `check` group, which must be explicitly enabled to display its messages.

- A single message identifier (within parentheses or not). Use the `-verbose` option on the `cc` command to obtain the message identifier.
- The name of a single message group (within parentheses, or not). Message-group names are:

  `all`

  All the messages in the compiler.

  `alignment`

  Messages about unusual or inefficient data alignment.

  `c_to_cxx`

  Messages reporting the use of C features that would be invalid or have a different meaning if compiled by a C++ compiler.

  `check`

  Messages reporting code or practices that, although correct and perhaps portable, are sometimes considered ill-advised because they can be confusing or fragile to maintain. For example, assignment as the test expression in an `if` statement. The `check` group gets defined by enabling `level5` messages.

  `nonansi`

  Messages reporting the use of non-ANSI Standard features.

  `defunct`

  Messages reporting the use of obsolete features: ones that were commonly accepted by early C compilers but were subsequently removed from the language.

  `obsolescent`

  Messages reporting the use of features that are valid in ANSI Standard C, but which were identified in the standard as being obsolete and likely to be removed from the language in a future version of the standard.

`overflow`

Messages that report assignments or casts that can cause overflow or other loss of data significance.

`performance`

Messages reporting code that might result in poor run-time performance.

`portable`

Messages reporting the use of language extensions or other constructs that might not be portable to other compilers or platforms.

`preprocessor`

Messages reporting questionable or nonportable use of preprocessing constructs.

`questcode`

Messages reporting questionable coding practices. Similar to the `check` group, but messages in this group are more likely to indicate a programming error rather than just a nonrobust style.

`returnchecks`

Messages related to function return values.

`uninit`

Messages related to using uninitialized variables.

`unused`

Messages reporting expressions, declarations, header files, static functions, and code paths that are not used.

- A single message-level name (within parentheses or not). Message-level names are:

  `level1`

  Important messages. These are less important than the level 0 core messages, because messages in this group are not displayed if `#pragma nostandard` is active.

`level2`

Moderately important messages. Level2 is the default for Compaq C.

`level3`

Less important messages.

`level4`

Useful check/portable messages.

`level5`

Not so useful check/portable messages.

`level6`

Additional noisy messages.

Be aware that there is a core of very important compiler messages that are enabled by default, regardless of what you specify with `#pragma message`. Referred to as message level0, it includes all messages issued in header files, and comprises what is known as the `nostandard` group. All other message levels add additional messages to this core of enabled messages.

You cannot modify `level0` (You cannot disable it, enable it, change its severity, or change its `emit_once` characteristic). However, you can modify individual messages in `level0`, provided such modification is allowed by the action. For example, you can disable a Warning or Informational in `level0`, or you can change an error in `level0` to a Fatal, and so on. (See restrictions on modifying individual messages.) Enabling a level also enables all the messages in the levels lower than it. So enabling `level3` messages also enables messages in `level2` and level1. Disabling a level also disables all the messages in the levels higher than it. So disabling `level4` messages also disables messages in `level5` and `level6`.

- A comma-separated list of message identifiers, group names, and message levels, freely mixed, enclosed in parentheses.

3.9.2 #pragma message option2

This form of the `#pragma message` directive has the following format:

`#pragma message` *`option2`*

The *`option2`* parameter must be one of the following keywords:

`save`

Saves the current state of which messages are enabled and disabled.

`restore`

Restores the previous state of which messages are enabled and disabled.

The `save` and `restore` options are useful primarily within header files.

3.9.3 #pragma message (“string”)

This form of the `#pragma message` directive is provided for compatibility with the Microsoft `#pragma message` directive, and it has the following format:

`#pragma message ("`*`string`*`")`

The directive emits the specified *`string`* as a compiler message. For example, when the compiler encounters the following line in the source file:

```
#pragma message ("hello")
```

It emits:

```
cc: Info: a.c, line 10: hello (simplemessage)
#pragma message ("hello")
----------------^
```

This form of the pragma is subject to macro replacement. For example, the following is allowed:

```
#pragma message ("Compiling file " __FILE__)
```

3.10 The #pragma optimize Directive

The `#pragma optimize` directive sets the optimization characteristics of function definitions that follow the directive. It allows optimization-control options that are normally set on the command line for the entire compilation to be specified in the source file for individual functions. This pragma has the following formats:

`#pragma optimize` *`settings`*

`#pragma optimize save`

`#pragma optimize restore`

`#pragma optimize command_line`

The `save` and `restore` options save and restore the current optimization state (level, unroll count, ansi-alias setting, and intrinsic setting).

The `command_line` option causes the optimization settings to revert to what was in effect at the start of the compilation, as specified by the `cc` command line options.

The *`settings`* are any combination of the following:

`level`

The `level` setting establishes the optimization level. Specify the level as follows:

```
level=n
```

Where *`n`* is an integer from 0 to 5:

`0` Disables all optimizations. Does not check for unassigned variables.

`1` Enables local optimizations and recognition of some common subexpressions. The callgraph determines the order of compilation of procedures.

`2` Includes `level 1` optimizations. Enables global optimization. This includes data-flow analysis, code motion, strength reduction and test replacement, split lifetime analysis, and code scheduling.

`3` Includes `level 2` optimizations. Enables additional global optimizations that improve speed (at the cost of extra code size), for example: integer multiplication and division expansion (using shifts), loop unrolling, and code replication to eliminate branches.

`4` Includes `level 3` optimizations. Enables interprocedural analysis and automatic inlining of small procedures (with heuristics limiting the amount of extra code). This is the default.

`5` Includes `level 4` optimizations. Activates software pipelining, which is a specialized form of loop unrolling that in certain cases improves run-time performance. Software pipelining uses instruction scheduling to eliminate instruction stalls within loops, rearranging instructions between different unrolled loop iterations to improve performance.

Loops chosen for software pipelining are always innermost loops and do not contain branches or procedure calls. To determine whether using `level 5` benefits your particular program, you should time program execution for the same program compiled at `level 4` and `level 5`. For programs that contain loops that exhaust available registers, longer execution times may result with `level 5`.

`unroll`

The `unroll` setting controls loop unrolling. Specify as follows:

```
unroll=n
```

Where *n* is a nonnegative integer. `unroll=` *n* means to unroll loop bodies *n* times, where *n* is between `0` and `16`. `unroll=0` means the optimizer will use its own default unroll amount. Specify `unroll` only at level `3` or higher.

`ansi-alias`

The `ansi-alias` setting controls ansi-alias assumptions. Specify one of the following:

```
ansi_alias=on
ansi_alias=off
```

`intrinsic`

The `intrinsic` setting controls recognition of intrinsics. Specify on of the following:

```
intrinsics=on
intrinsics=off
```

Whitespace is optional between the setting clauses and before and after the "=" in each clause. The `pragma` is not subject to macro replacement.

Example:

```
#pragma optimize level=5 unroll=6
```

Usage Notes

- If the `level=0` clause is present, it must be the only clause present.
- The `#pragma optimize` directive must appear at file scope, outside any function body.

- The #pragma environment command_line directive resets the optimization state to that specified on the command line.
- If #pragma optimize does not specify a setting for one of the optimization states, that state remains unchanged.
- When a function definition is encountered, it is compiled using the optimization settings that are current at that point in the source.
- When a function is compiled under level=0, the compiler will not inline that function. In general, when functions are inlined, the inlined code is optimized using the optimization controls in effect at the call site instead of using the optimization controls specified for the function being inlined.

3.11 The #pragma pack Directive

The #pragma pack directive changes the alignment restrictions on all members of a structure. The pragma must appear before the entire structure definition because it acts on the whole structure. The syntax of this pragma is as follows:

#pragma pack {*n* **| (***n***) | ( )}**

The *n* is a number (such as 1, 2, or 4) that specifies that subsequent structure members are to be aligned on *n*-byte boundaries. If you supply a value of 0 (zero) for *n*, the alignment reverts to the default, which may have been set by the -Zp*n* option on the cc command.

The pragma pack directive also supports the push and pop arguments established by Microsoft Visual C++:

#pragma pack ({push | pop} [**,** ***identifier***] [**,** ***n***]**)**

With push and pop arguments, you can save and restore alignment values across program components. This allows you to combine components into a single translation unit even if they specify different alignments:

- Every occurrence of pragma pack with a push argument stores the current packing alignment value on an internal compiler stack. If you provide a value for *n*, that value becomes the new alignment value. If you specify an *identifier*, a name of your choosing, it is associated with the new value.
- Every occurrence of pragma pack with a pop argument retrieves the value at the top of the stack and makes that value the new alignment value. If an empty stack is popped, the alignment value reverts to the default alignment, and a warning is issued. If you specify a value for *n*, that value becomes the new alignment value.

If you specify an *identifier*, all alignment values stored on the stack are removed from the stack until a matching *identifier* is found. The value associated with the *identifier* is also removed from the stack, and the value that was in effect just before the *identifier* was pushed becomes the new alignment value. If no matching *identifier* is found, the alignment value reverts to its default value, and a warning is issued.

The push and pop arguments of pragma pack allow you to write header files that ensure that alignment values are the same before and after the header file is encountered. For example:

```
/* File name: myinclude.h */
#pragma pack (push,enter_myinclude)
.
.  (your include file code)
.
#pragma pack (pop,enter_myinclude)
/* End of myinclude.h */
```

In this example, the current alignment value is associated with the identifier enter_myinclude and pushed on entry to the header file. Then the code in your include file is processed. After the include file processing is completed, the #pragma pack at the end of the header file removes all intervening alignment values that might have occurred in the header file, as well as the alignment value associated with enter_myinclude, thereby preserving the same alignment value before and after the header file.

The push and pop arguments also allow you to include header files that might set alignment values that differ from the values set in your code. For example:

```
#pragma pack (push,before_myinclude)
#include <myinclude.h>
#pragma pack (pop,before_myinclude)
```

In this example, your code is protected from any changes to the alignment value that might occur in myinclude.h by saving the current alignment value, processing the include file (which may leave the alignment value with an unknown setting), and restoring the original alignment value.

3.12 The #pragma pointer_size Directive

The #pragma pointer_size directive controls pointer size allocation for the following:

- References
- Pointer declarations
- Function declarations
- Array declarations

This pragma has the following syntax:

#pragma pointer_size { long | short | 64 | 32 } | { save | restore }

long | 64

Sets all pointer sizes as 64 bits in all declarations that follow this directive, until the compiler encounters another #pragma pointer_size directive.

short | 32

Sets all pointer sizes as 32 bits in declarations that follow this directive, until the compiler encounters another #pragma pointer_size directive.

save | restore

Saves the current pointer size and restores the saved pointer size, respectively. The save and restore options are particularly useful for specifying mixed pointer support and for protecting header files that interface to older objects. Objects compiled with multiple pointer size pragmas will not be compatible with old objects, and the compiler cannot discern that incompatible objects are being mixed.

For example:

```
#pragma pointer_size long
  /* pointer sizes in here are 64-bits */
#pragma pointer_size save
#pragma pointer_size short
  /* pointer sizes in here are 32-bits */
#pragma pointer_size restore
  /* pointer sizes in here are again 64-bits */
```

The use of short pointers is restricted to Compaq C++ and Compaq C compilers residing on a Tru64 UNIX system. Programs should not attempt to pass short pointers from C++ routines to routines written in any language other than the C programming language. Also, Compaq C++ may require explicit conversion of short pointers to long pointers in applications that use short pointers. You should first port those applications in which you are considering using short pointers, and then analyze them to determine if short pointers would be beneficial. A difference in the size of a pointer in a function declaration is not sufficient to overload a function.

The C compiler issues an error level diagnostic if it encounters any of the following conditions:

- Two functions defined differ only with respect to pointer sizes.
- Two functions differ in return type only with respect to pointer size.

3.13 The #pragma unroll Directive

The #pragma unroll directive controls the amount of loop unrolling performed on a subsequent for loop. The directive has the following format:

#pragma unroll (*unroll-factor***)**

The directive directs the compiler to unroll the for loop that follows it by the number of times specified by the *unroll-factor* argument.

unroll-factor

An integer constant in the range from 0 to 255 that suggests the number of times the compiler should unroll the for loop that follows the directive. A value of 0 will cause the directive to be ignored, and the compiler will determine the number of times to unroll the loop in its normal way. A value of 1 will prevent the loop from being unrolled.

The directive must be immediately followed by the for statement that it is to control; otherwise a warning is issued and the pragma is ignored.

Example

```
#pragma unroll (1)
for (i=0; i<1000; i++) {foo(i);}
```

3.14 The #pragma use_linkage Directive

After defining a special linkage with the #pragma linkage directive, as described in Section 3.7, use the #pragma use_linkage directive to associate the linkage with a function.

This pragma has the following syntax:

#pragma use_linkage *linkage-name* **(***id1, id2, ...***)**

linkage-name

Specifies the name of a linkage that the #pragma linkage directive previously defined.

id1, *id2*, *...*

Specifies the names of functions, or typedef names of function type, that you want to associate with the specified linkage.

If you specify a typedef name of function type, then functions or pointers to functions that are declared using that type will have the specified linkage.

The #pragma use_linkage directive must appear in the source file before any use or definition of the specified routines. Otherwise, the results are unpredictable.

The following example defines a special linkage and associates it with a routine that takes three integer parameters and returns a single integer result in the same location where the first parameter was passed:

```
#pragma linkage example_linkage (parameters(r16, r17, r19), result(r16))
#pragma use_linkage example_linkage (sub)
int sub (int p1, int p2, short p3);

main()
{
    int result;

    result = sub (1, 2, 3);
}
```

In this example, the `result(r16)` option indicates that the function result will be returned in register `r16` instead of the usual location (`r0`). The `parameters` option indicates that the three parameters that are passed to `sub` should be passed in registers `r16`, `r17`, and `r19`.

In the following example, both the function `f1` and the function type `t` are given the linkage `foo`. The invocation through the function pointer `f2` will correctly invoke the function `f1` using the special linkage.:

```
#pragma linkage foo = (parameters(r1), result(r4))
#pragma use_linkage foo(f1,t)

int f1(int a);
typedef int t(int a);

t *f2;

#include <stdio.h>

main() {
    f2 = f1;
    b = (*f2)(1);
}
```

3.15 The #pragma weak Directive

The `#pragma weak` directive defines a new weak external symbol and associates this new symbol with an external symbol. The syntax for this pragma is as follows:

`#pragma weak (`*`secondary-name, primary-name`***`)`**

See Section 2.8 for information on strong and weak symbols.

4

Shared Libraries

Shared libraries are the default system libraries. The default behavior of the C compiler is to use shared libraries when performing compile and link operations.

This chapter addresses the following topics:

- Overview of shared libraries (Section 4.1)
- Resolving symbols (Section 4.2)
- Linking with shared libraries (Section 4.3)
- Turning off shared libraries (Section 4.4)
- Creating shared libraries (Section 4.5)
- Working with private shared libraries (Section 4.6)
- Using quickstart (Section 4.7)
- Debugging programs linked with shared libraries (Section 4.8)
- Loading a shared library at run time (Section 4.9)
- Protecting shared library files (Section 4.10)
- Shared library versioning (Section 4.11)
- Symbol binding (Section 4.12)
- Shared library restrictions (Section 4.13)

4.1 Shared Library Overview

Shared libraries consist of executable code that can be located at any available address in memory. Only one copy of a shared library's instructions is loaded, and the system shares that one copy among multiple programs instead of loading a copy for each program using the library, as is the case with archive (static) libraries.

Programs that use shared libraries enjoy the following significant advantages over programs that use archive libraries:

- Programs linked with shared libraries do not need to be recompiled and relinked when changes are made to those libraries.

- Unlike programs linked with archive libraries, programs linked with shared libraries do not include library routines in the executable program file. Programs linked with shared libraries include information to load the shared library and gain access to its routines and data at load time.

  This means that use of shared libraries occupies less space in memory and on disk. When multiple programs are linked to a single shared library, the amount of physical memory used by each process can be significantly reduced.

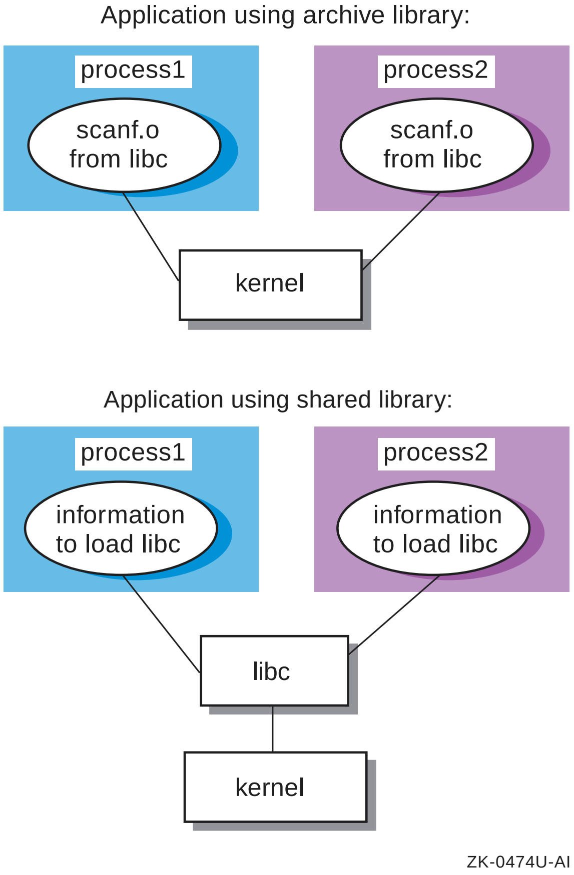
From a user perspective, the use of shared libraries is transparent. In addition, you can build your own shared libraries and make them available to other users. Most object files and archive libraries can be made into shared libraries. See Section 4.5 for more information on which files can be made into shared libraries.

Shared libraries differ from archive libraries in the following ways:

- You build shared libraries by using the `ld` command with the appropriate options. You create archive libraries by using the `ar` command. For more information on the `ld` command, see `ld`(1).
- When shared libraries are linked into an executable program, they can be positioned at any available address. At run time, the loader (`/sbin/loader`) assigns a location in the process's private virtual address space. In contrast, when archive libraries are linked into an executable program, they have a fixed location in the process's private virtual address space.
- Shared libraries reside in the `/usr/shlib` directory. Archive libraries reside in the `/usr/lib` directory.
- Shared library naming convention specifies that a shared library name begins with the prefix `lib` and ends with the suffix `.so`. For example, the library containing common C language functions is `libc.so`. Archive library names also begin with the prefix `lib`, but they end with the suffix `.a`.

Figure 4–1 shows the difference between the use of archive and shared libraries.

Figure 4–1: Use of Archive and Shared Libraries



4.2 Resolving Symbols

Symbol resolution is the process of mapping an unresolved symbol imported by a program or shared library to the pathname of the shared library that exports that symbol. Symbols are resolved in much the same way for shared and archive libraries, except that the final resolution of symbols in shared objects does not occur until a program is invoked.

The following sections describe:

- Search path of the linker (`ld`) (Section 4.2.1)
- Search path of the run-time loader (`/sbin/loader`) (Section 4.2.2)
- Name resolution (Section 4.2.3)
- Options to the `ld` command to determine how unresolved external symbols are to be handled (Section 4.2.4)

4.2.1 Search Path of the Linker

When the linker (`ld`) searches for files that have been specified by using the `-l` option on the command line, it searches each directory in the order shown in the following list, looking first in each directory for a shared library (`.so`) file:

1. `/usr/shlib`
2. `/usr/ccs/lib`
3. `/usr/lib/cmplrs/cc`
4. `/usr/lib`
5. `/usr/local/lib`
6. `/var/shlib`

If the linker does not find a shared library, it searches through the same directories again, looking for an archive (`.a`) library. You can prevent the search for archive libraries by using the `–no_archive` option on the `ld` command.

4.2.2 Search Path of the Run-time Loader

Unless otherwise directed, the run-time loader (`/sbin/loader`) follows the same search path as the linker. You can use one of the following methods to direct the run-time loader to look in directories other than those specified by the default search path:

- Specify a directory path by using the `-rpath` *`string`* option to the `ld` command and setting *`string`* to the list of directories to be searched.
- Set the environment variable `LD_LIBRARY_PATH` to point to the directory in which you keep your private shared libraries before executing your programs. The run-time loader examines this variable when the program is executed; if it is set, the loader searches the paths defined by `LD_LIBRARY_PATH` before searching the list of directories discussed in Section 4.2.1.

  You can set the `LD_LIBRARY_PATH` variable by either of the following methods:

  – Set it as an environment variable at the shell prompt.

    For the C shell, use the `setenv` command followed by a colon-separated path. For example:

    ```
    % setenv LD_LIBRARY_PATH .:$HOME/testdir
    ```

    For the Bourne and Korn shells, set the variable and then export it. For example:

```
$ LD_LIBRARY_PATH=.:$HOME/testdir
$ export LD_LIBRARY_PATH
```

These examples set the path so that the loader looks first in the current directory and then in your $HOME/testdir directory.

- Add the definition of the variable to your login or shell startup files. For example, you could add the following line to your .login or .cshrc file if you work in the C shell:

```
setenv LD_LIBRARY_PATH .:$HOME/testdir:/usr/shlib
```

If the loader cannot find the library it needs in the paths defined by any of the preceding steps, it looks through the directories specified in the default path described in Section 4.2.1. In addition, you can use the _RLD_ROOT environment variable to alter the search path of the run-time loader. For more information, see loader(5).

4.2.3 Name Resolution

The semantics of symbol name resolution are based on the order in which the object file or shared object containing a given symbol appears on the link command line. The linker normally takes the leftmost definition for any symbol that must be resolved.

The sequence in which names are resolved proceeds as if the link command line was stored in the executable program. When the program runs, all symbols that are accessed during execution must be resolved. The loader aborts execution of the program if an unresolved text symbol is accessed.

For information on how unresolved symbols are handled by the system, see Section 4.2.4. The following sequence resolves references to any symbol from the main program or from a library:

1. If a symbol is defined in an object or in an archive library from which you build the main executable program file, that symbol is used by the main program file and all of the shared libraries that it uses.
2. If the symbol is not defined by the preceding step and is defined by one or more of the shared objects linked with the executable program, then the leftmost library on the link command line containing a definition is used.
3. If the libraries on the link command line were linked to be dependent on other libraries, then the dependencies of libraries are searched in a breadth-first fashion instead of being searched in a depth-first fashion. For example, as shown in the following diagram, executable program A is linked against shared library B and shared library D, and library B is linked against library C.

```
      A
     / \
    B   D
   /
  C
```

The search order is A-B-D-C. In a breadth-first search, the grandchildren of a node are searched after all the children have been searched.

4. If the symbol is not resolved in any of the previous steps, the symbol remains unresolved.

Note that because symbol resolution always prefers the main object, shared libraries can be set up to call back into a defined symbol in the main object. Likewise, the main object can define a symbol that will override (preempt or hook) a definition in a shared library.

4.2.4 Options to Determine Handling of Unresolved External Symbols

The default behavior of the linker when building executable programs differs from its default behavior when building shared libraries:

- Executable programs — When building executable programs, an unresolved symbol produces an error by default (`-error_unresolved` option). The link fails and the output file is not marked as executable.
- Shared libraries — When building shared libraries, an unresolved symbol produces only a warning message by default (`-warning_unresolved` option).

When building shared libraries and `-error_unresolved` is specified, an output file (`.so` file) is produced even if unresolved symbols are present, overwriting any pre-existing `.so` file with the same name. See the *Programmer's Guide* for information about how to control overwriting of linker output files.

You can control how the linker handles unresolved symbols by using the following options to the `ld` command:

- `-expect_unresolved` *`pattern`*
- `-warning_unresolved`
- `-error_unresolved`

See `ld`(1) for details about the effects of these options.

4.3 Linking with Shared Libraries

When compiling and linking a program, using shared libraries is the same as using static libraries. For example, the following command compiles program `hello.c` and links it against the default system C shared library `libc.so`:

```
% cc -o hello hello.c
```

You can pass certain `ld` command options to the `cc` command to allow flexibility in determining the search path for a shared library. For example, you can use the `-L`*`dir`* option with the `cc` command to change the search path by adding *`dir`* before the default directories, as shown in the following example:

```
% cc -o hello hello.c -L/usr/person -lmylib
```

To exclude the default directories from the search and limit the search to specific directories and specific libraries, specify the `-L` option first with no arguments. Then, specify it again with the directory to search, followed by the `-l` option with the name of the library to search for. For example, to limit the search path to `/usr/person` for use with the private library `libmylib.so`, enter the following command:

```
% cc -o hello hello.c -L -L/usr/person -lmylib
```

Note that because the `cc` command always implicitly links in the C library, the preceding example requires that a copy of `libc.so` or `libc.a` must be in the `/usr/person` directory.

4.4 Turning Off Shared Libraries

In application linking, the default behavior is to use shared libraries. To link an application that does not use shared libraries, you must use the `-non_shared` option to the `cc` or `ld` commands when you link that application.

For example:

```
% cc -non_shared -o hello hello.c
```

Although shared libraries are the default for most programming applications, some applications cannot use shared libraries:

- Applications that need to run in single-user mode cannot be linked with shared libraries because the `/usr/shlib` directory must be mounted to provide access to shared libraries.
- Applications whose sole purpose is single-user benchmarks should not be linked with shared libraries.

4.5 Creating Shared Libraries

You create shared libraries by using the ld command with the -shared option. You can create shared libraries from object files or from existing archive libraries.

4.5.1 Creating Shared Libraries from Object Files

To create the shared library libbig.so from the object files bigmod1.o and bigmod2.o, enter the following command:

```
% ld -shared -no_archive -o libbig.so bigmod1.o bigmod2.o -lc
```

The -no_archive option tells the linker to resolve symbols using only shared libraries. The -lc option tells the linker to look in the system C shared library for unresolved symbols.

To make a shared library available on a system level by copying it into the /usr/shlib directory, you must have root privileges. System shared libraries should be located in the /usr/shlib directory or in one of the default directories so that the run-time loader (/sbin/loader) can locate them without requiring every user to set the LD_LIBRARY_PATH variable to directories other than those in the default path.

4.5.2 Creating Shared Libraries from Archive Libraries

You can also create a shared library from an existing archive library by using the ld command. The following example shows how to convert the static library old.a into the shared library libold.so:

```
% ld -shared -no_archive -o libold.so -all old.a -none -lc
```

In this example, the -all option tells the linker to link all the objects from the archive library old.a. The -none option tells the linker to turn off the -all option. Note that the -no_archive option applies to the resolution of the -lc option but not to old.a (because old.a is explicitly mentioned).

4.6 Working with Private Shared Libraries

In addition to system shared libraries, any user can create and use private shared libraries. For example, you have three applications that share some common code. These applications are named user, db, and admin. You decide to build a common shared library, libcommon.so, containing all the symbols defined in the shared files io_util.c, defines.c, and network.c. To do this, follow these steps:

1. Compile each C file that will be part of the library:

```
% cc -c io_util.c
% cc -c defines.c
% cc -c network.c
```

2. Create the shared library `libcommon.so` by using the `ld` command:

   ```
   % ld -shared -no_archive \
   -o libcommon.so io_util.o defines.o network.o -lc
   ```

3. Compile each C file that will be part of the application:

   ```
   % cc -c user.c
   % cc -o user user.o -L. -lcommon
   ```

   Note that the second command in this step tells the linker to look in the current directory and use the library `libcommon.so`. Compile `db.c` and `admin.c` in the same manner:

   ```
   % cc -c db.c
   % cc -o db db.o -L. -lcommon

   % cc -c admin.c
   % cc -o admin admin.o -L. -lcommon
   ```

4. Copy `libcommon.so` into a directory pointed to by `LD_LIBRARY_PATH`, if it is not already in that directory.

5. Run each compiled program (`user`, `db`, and `admin`).

4.7 Using Quickstart

One advantage of using shared libraries is the ability to change a library after all executable images have been linked and to fix bugs in the library. This ability is very useful during the development phase of an application.

During the production cycle, however, the shared libraries and applications that you develop are often fixed and will not change until the next release. If this is the case, you can take advantage of quickstart, a method of using predetermined addresses for all symbols in your program and libraries.

No special link options are required to prepare an application for quickstarting; however, a certain set of conditions must be satisfied. If an object cannot be quickstarted, it still runs but startup time is slower.

When the linker creates a shared object (a shared library or a main executable program that uses shared libraries), it assigns addresses to the text and data portions of the object. These addresses are what might be called quickstarted addresses. The linker performs all dynamic relocations in advance, as if the object will be loaded at its quickstarted address.

Any object depended upon is assumed to be at its quickstarted address. References to that object from the original object have the address of the depended-upon object set accordingly.

To use quickstart, an object must meet the following conditions:

- The object's actual run-time memory location must match the quickstart location. The run-time loader tries to use the quickstart location. However, if another library is already occupying that spot, the object will not be able to use it.
- All depended-upon objects must be quickstarted.
- All depended-upon objects must be unchanged since they were linked. If objects have changed, addresses of functions within the library might have moved or new symbols might have been introduced that can affect the loading. (Note that you might still be able to quickstart objects that have been modified since linking by running the `fixso` utility on the changed objects. See `fixso`(1) for more information.)

The operating system detects these conditions by using checksums and timestamps.

When you build libraries, they are given a quickstart address. Unless each library used by an application chooses a unique quickstart address, the quickstart constraints cannot be satisfied. Rather than worry about addresses on an application basis, give a unique quickstart address to each shared library that you build to ensure that all of your objects can be loaded at their quickstart addresses.

The linker maintains the `so_locations` database to register each quickstart address when you build a library. The linker avoids addresses already in the file when choosing a quickstart address for a new library.

By default, `ld` runs as though the `-update_registry ./so_locations` option has been selected, so the `so_locations` file in the directory of the build is updated (or created) as necessary.

To ensure that your libraries do not collide with shared libraries on your system, enter the following commands:

```
% cd <directory_of_build>
% cp /usr/shlib/so_locations .
% chmod +w so_locations
```

You can now build your libraries. If your library builds occur in multiple directories, use the `-update_registry` option to the `ld` command to explicitly specify the location of a common `so_locations` file. For example:

```
% ld -shared -update_registry /common/directory/so_locations ...
```

If you install your shared libraries globally for all users of your system, update the systemwide `so_locations` file. Enter the following commands as root, with *`shared_library`*`.so` being the name of your actual shared library:

```
# cp shared_library.so /usr/shlib
# mv /usr/shlib/so_locations /usr/shlib/so_locations.old
# cp so_locations /usr/shlib
```

If several people are building shared libraries, the common so_locations file must be administered as any shared database would be. Each shared library used by any given process must be given a unique quickstart address in the file. The range of default starting addresses that the linker assigns to main executable files does not conflict with the quickstarted addresses it creates for shared objects. Because only one main executable file is loaded into a process, an address conflict never occurs between a main file and its shared objects.

If you are building only against existing shared libraries (and not building your own libraries), you do not need to do anything special. As long as the libraries meet the previously described conditions, your program will be quickstarted unless the libraries themselves are not quickstarted. Most shared libraries shipped with the operating system are quickstarted.

If you are building shared libraries, you must first copy the so_locations file as previously described. Next, you must build all shared libraries in bottom-up dependency order, using the so_locations file. Specify all depended-upon libraries on the link line. After all the libraries are built, you can then build your applications.

4.7.1 Verifying That an Object Is Quickstarting

To test whether an application's executable program is quickstarting, set the _RLD_ARGS environment variable to −quickstart_only and run the program. For example:

```
% setenv _RLD_ARGS -quickstart_only
% foo
(non-quickstart output)
21887:foo: /sbin/loader: Fatal Error: NON-QUICKSTART detected \
  -- QUICKSTART must be enforced
```

If the program runs successfully, it is quickstarting. If a load error message is produced, the program is not quickstarting.

4.7.2 Manually Tracking Down Quickstart Problems

To determine why an executable program is not quickstarting, you can use the fixso utility, described in Section 4.7.3, or you can manually test for the conditions described in the following list of requirements. Using fixso is easier, but it is helpful to understand the process involved.

1. The executable program must be able to be quickstarted.

Test the quickstart flag in the dynamic header. The value of the quickstart flag is 0x00000001. For example:

```
% odump -D foo | grep FLAGS
```

(non-quickstart output)

```
  FLAGS: 0x00000000
```

(quickstart output)

```
  FLAGS: 0x00000001
```

If the quickstart flag is not set, one or more of the following conditions exists:

- The executable program was linked with unresolved symbols. Make sure that the `ld` options `-warning_unresolved` and `-expect_unresolved` are not used when the executable program is linked. Fix any unresolved symbol errors that occur when the executable program is linked.
- The executable program is not linked directly against all of the libraries that it uses at run time. Add the option `-transitive_link` to the `ld` options used when the executable program is built.

2. The executable program's dependencies must be able to be quickstarted. Get a list of an executable program's dependencies. For example:

```
% odump -Dl foo
```

(quickstart output)

```
                 ***LIBRARY LIST SECTION***
  Name             Time-Stamp        CheckSum   Flags Version
foo:
  libX11.so    Sep 17 00:51:19 1993 0x78c81c78  NONE
  libc.so      Sep 16 22:29:50 1993 0xba22309c  NONE osf.1
  libdnet_stub.so Sep 16 22:56:51 1993 0x1d568a0c  NONE osf.1
```

Test the quickstart flag in the dynamic header of each of the dependencies:

```
% cd /usr/shlib
% odump -D libX11.so libc.so libdnet_stub.so | grep FLAGS
```

(quickstart output)

```
  FLAGS: 0x00000001
  FLAGS: 0x00000001
  FLAGS: 0x00000001
```

If any of these dependencies cannot be quickstarted, the same measures suggested in step 1 can be applied here, provided that the shared library can be rebuilt by the user.

3. The timestamp and checksum information must match for all dependencies.

   The dependencies list in step 2 shows the expected values of the timestamp and checksum fields for each of foo's dependencies. Match these values against the current values for each of the libraries:

   ```
   % cd /usr/shlib
   % odump -D libX11.so libc.so libdnet_stub.so | \
   grep TIME_STAMP
   ```

   (quickstart output)

   ```
     TIME_STAMP: (0x2c994247) Fri Sep 17 00:51:19 1993
     TIME_STAMP: (0x2c99211e) Thu Sep 16 22:29:50 1993
     TIME_STAMP: (0x2c992773) Thu Sep 16 22:56:51 1993
   ```

   ```
   % odump -D libX11.so libc.so libdnet_stub.so | grep CHECKSUM
   ```

   (quickstart output)

   ```
     ICHECKSUM: 0x78c81c78
     ICHECKSUM: 0xba22309c
     ICHECKSUM: 0x1d568a0c
   ```

   If any of the tests in these examples shows a timestamp or checksum mismatch, relinking the program should fix the problem.

   You can use the version field to verify that you have identified the correct libraries to be loaded at run time. To test the dependency versions, use the odump command as shown in the following example:

   ```
   % odump -D libX11.so | grep IVERSION
   % odump -D libc.so | grep IVERSION
     IVERSION: osf.1
   % odump -D libdnet_stub.so | grep IVERSION
     IVERSION: osf.1
   ```

   The lack of an IVERSION entry is equivalent to a blank entry in the dependency information. It is also equivalent to the special version _null.

   If any version mismatches are identified, you can normally find the correct matching version of the shared library by appending the version identifier from the dependency list or _null to the path /usr/shlib.

4. Each of the executable program's dependencies must also contain dependency lists with matching timestamp and checksum information.

   Repeat step 3 for each of the shared libraries in the executable program's list of dependencies:

   ```
   % odump -Dl libX11.so
   ```

   (quickstart output)

```
                  ***LIBRARY LIST SECTION***
  Name             Time-Stamp        CheckSum   Flags Version
libX11.so:
  libdnet_stub.so Sep 16 22:56:51 1993 0x1d568a0c  NONE osf.1
  libc.so      Sep 16 22:29:50 1993 0xba22309c  NONE osf.1
% odump -D libdnet_stub.so libc.so | grep TIME_STAMP
  TIME_STAMP: (0x2c992773) Thu Sep 16 22:56:51 1993
  TIME_STAMP: (0x2c99211e) Thu Sep 16 22:29:50 1993
% odump -D libdnet_stub.so libc.so | grep CHECKSUM
  ICHECKSUM: 0x1d568a0c
  ICHECKSUM: 0xba22309c
```

If the timestamp or checksum information does not match, the shared library must be rebuilt to correct the problem. Rebuilding a shared library will change its timestamp and, sometimes, its checksum. Rebuild dependencies in bottom-up order so that an executable program or shared library is rebuilt after its dependencies have been rebuilt.

4.7.3 Tracking Down Quickstart Problems with the fixso Utility

The `fixso` utility can identify and repair quickstart problems caused by timestamp and checksum discrepancies. It can repair programs as well as the shared libraries they depend on, but it might not be able to repair certain programs, depending on the degree of symbolic changes required.

The `fixso` utility cannot repair a program or shared library if any of the following restrictions apply:

- The program or shared library depends on other shared libraries that cannot be quickstarted. This restriction can be avoided by using `fixso` to repair shared libraries in bottom-up order.
- New name conflicts are introduced after a program or shared library is created. Name conflicts result when the same global symbol name is exported by two or more shared library dependencies or by the program and one of its shared library dependencies.
- The program's shared library dependencies are not all loaded at their quickstart locations. A shared library cannot be loaded at its quickstart locations if other shared libraries are loaded at that location and are already in use. This rule applies systemwide, not just to individual processes. To avoid this restriction, use a common `so_locations` file for registering unique addresses for shared libraries.
- The program or shared library depends on an incompatible version of another shared library. This restriction can be avoided by instructing `fixso` where to find a compatible version of the offending shared library.

The `fixso` utility can identify quickstart problems as shown in the following example:

```
% fixso -n hello.so
fixso: Warning: found ’/usr/shlib/libc.so’ (0x2d93b353) which does
       not match timestamp 0x2d6ae076 in liblist of hello.so, will fix
fixso: Warning: found ’/usr/shlib/libc.so’ (0xc777ff16) which does
       not match checksum 0x70e62eeb in liblist of hello.so, will fix
```

The `-n` option suppresses the generation of an output file. Discrepancies are reported, but `fixso` does not attempt to repair the problems it finds. The following example shows how you can use `fixso` to repair quickstart problems:

```
% fixso -o ./fixed/main main
fixso: Warning: found ’/usr/shlib/libc.so’ (0x2d93b353) which does
       not match timestamp 0x2d7149c9 in liblist of main, will fix
% chmod +x fixed/main
```

The `-o` option specifies an output file. If no output file is specified, `fixso` uses `a.out`. Note that `fixso` does not create the output file with execute permission. The `chmod` command allows the output file to be executed. This change is necessary only for executable programs and can be bypassed when using `fixso` to repair shared libraries.

If a program or shared library does not require any modifications to repair quickstart, `fixso` indicates this as shown in the following example:

```
% fixso -n /bin/ls
no fixup needed for /bin/ls
```

4.8 Debugging Programs Linked with Shared Libraries

Debugging a program that uses shared libraries is essentially the same as debugging a program that uses archive libraries.

The `dbx` debugger’s `listobj` command displays the names of the executable programs and all of the shared libraries that are known to the debugger. See Chapter 5 for more information about using `dbx`.

4.9 Loading a Shared Library at Run Time

In some situations, you might want to load a shared library from within a program. This section includes two short C program examples and a makefile to demonstrate how to load a shared library at run time.

The following example (`pr.c`) shows a C source file that prints out a simple message:

```
printmsg()
    {
        printf("Hello world from printmsg!\n");
    }
```

The next example (used1.c) defines symbols and demonstrates how to use the dlopen function:

```
#include <stdio.h>
#include <dlfcn.h>

/* All errors from dl* routines are returned as NULL */
#define BAD(x)          ((x) == NULL)

main(int argc, char *argv[])
{
    void *handle;
    void (*fp)();

    /* Using "./" prefix forces dlopen to look only in the current
     * current directory for pr.so.  Otherwise, if pr.so was not
     * found in the current directory, dlopen would use rpath,
     * LD_LIBRARY_PATH and default directories for locating pr.so.
     */
    handle = dlopen("./pr.so", RTLD_LAZY);
    if (!BAD(handle)) {
        fp = dlsym(handle, "printmsg");
        if (!BAD(fp)) {
            /*
             * Here is where the function
             * we just looked up is called.
             */
            (*fp)();
        }
        else {
            perror("dlsym");
            fprintf(stderr, "%s\n", dlerror());
        }
    }
    else {
        perror("dlopen");
        fprintf(stderr, "%s\n", dlerror());
    }
    dlclose(handle);
}
```

The following example shows the makefile that makes pr.o, pr.so, so_locations, and usedl.o:

```
# this is the makefile to test the examples

all:    runit

runit:  usedl pr.so
        ./usedl

usedl:  usedl.c
        $(CC) -o usedl usedl.c

pr.so:  pr.o
        $(LD) -o pr.so -shared pr.o -lc
```

4.10 Protecting Shared Library Files

Because of the sharing mechanism used for shared libraries, normal file system protections do not protect libraries against unauthorized reading. For example, when a shared library is used in a program, the text part of that library can be read by other processes even when the following conditions exist:

- The library's permissions are set to 600.
- The other processes do not own the library or are not running with their UID set to the owner of that library.

Only the text part of the library, not the data segment, is shared in this manner.

To prevent unwanted sharing, link any shared libraries that need to be protected by using the linker's `-T` and `-D` options to put the data section in the same 8-MB segment as the text section. For example, enter a command similar to the following:

```
% ld -shared -o libfoo.so -T 30000000000  \
-D 30000400000 object_files
```

In addition, segment sharing can occur with any file that uses the `mmap` system call without the `PROT_WRITE` flag as long as the mapped address falls in the same memory segment as other files using `mmap`.

Any program using `mmap` to examine files that might be highly protected can ensure that no segment sharing takes place by introducing a writable page into the segment before or during the `mmap`. The easiest way to provide protection is to use the `mmap` system call on the file with `PROT_WRITE` enabled in the protection, and use the `mprotect` system call to make the mapped memory read-only. Alternatively, to disable all segmentation and to avoid any unauthorized sharing, enter the following line in the configuration file:

```
segmentation 0
```

4.11 Shared Library Versioning

One of the advantages of using shared libraries is that a program linked with a shared library does not need to be rebuilt when changes are made to that library. When a changed shared library is installed, applications should work as well with the newer library as they did with the older one.

Note

Because of the need for address fixing, it can take longer to load an existing application that uses an older version of a shared library when a new version of that shared library is installed. You can avoid this kind of problem by relinking the application with the new library.

4.11.1 Binary Incompatible Modifications

Infrequently, a shared library might be changed in a way that makes it incompatible with applications that were linked with it before the change. This type of change is referred to as a binary incompatibility. A binary incompatibility introduced in a new version of a shared library does not necessarily cause applications that rely on the old version to break (that is, violate the backward compatibility of the library). The system provides shared library versioning to allow you to take steps to maintain a shared library's backward compatibility when introducing a binary incompatibility in the library.

Among the types of incompatible changes that might occur in shared libraries are the following:

- Removal of documented interfaces

  For example, if the `malloc()` function in `libc.so` was replaced with a function called (`__malloc`), programs that depend on the older function would fail due to the missing `malloc` symbol.

- Modification of documented interfaces

  For example, if a second argument to the `malloc()` function in `libc.so` was added, the new `malloc()` would probably fail when programs that depend on the older function pass in only one argument, leaving undefined values in the second argument.

- Modification of global data definitions

  For example, if the type of the `errno` symbol in `libc.so` was changed from an `int` to a `long`, programs linked with the older library might read and write 32-bit values to and from the newly expanded 64-bit data item. This might yield invalid error codes and indeterminate program behavior.

This is by no means an exhaustive list of the types of changes that result in binary incompatibilities. Shared library developers should exercise common sense to determine whether any change is likely to cause failures in applications linked with the library prior to the change.

4.11.2 Shared Library Versions

You can maintain the backward compatibility of a shared library affected by incompatible changes by providing multiple versions of the library. Each shared library is marked by a version identifier. You install the new version of the library in the library's default location, and the older, binary compatible version of the library in a subdirectory whose name matches that library's version identifier.

For example, if an incompatible change was made to `libc.so`, the new library (`/usr/shlib/libc.so`) must be accompanied by an instance of the library before the change (`/usr/shlib/osf.1/libc.so`). In this example, the older, binary compatible version of `libc.so` is the `osf.1` version. After the change is applied, the new `libc.so` is built with a new version identifier. Because a shared library's version identifier is listed in the shared library dependency record of a program that uses the library, the loader can identify which version of a shared library is required by an application (see Section 4.11.6).

In the example, a program built with the older `libc.so`, before the binary incompatible change, requires the `osf.1` version of the library. Because the version of `/usr/shlib/libc.so` does not match the one listed in the program's shared library dependency record, the loader will look for a matching version in `/usr/shlib/osf.1`.

Applications built after the incompatible change will use `/usr/shlib/libc.so` and will depend on the new version of the library. The loader will load these applications by using `/usr/shlib/libc.so` until some further binary incompatibility is introduced.

Table 4–1 describes the linker options used to effect version control of shared libraries.

Table 4–1: Linker Options That Control Shared Library Versioning

Option	Description
`-set_version` *`version-string`*	Establishes the version identifiers associated with a shared library. The string *`version-string`* is either a single version identifier or a colon-separated list of version identifiers. No restrictions are placed on the names of version identifiers; however, it is highly recommended that UNIX directory naming conventions be followed. If a shared library is built with this option, any program built against it will record a dependency on the specified version or, if a list of version identifiers is specified, the rightmost version specified in the list. If a shared library is built with a list of version identifiers, the run-time loader will allow any program to run that has a shared library dependency on any of the listed versions. This option is only useful when building a shared library (with `-shared`).
`-exact_version`	Sets an option in the dynamic object produced by the `ld` command that causes the run-time loader to ensure that the shared libraries the object uses at run time match the shared libraries used at link time. This option is used when building a dynamic executable file (with `-call_shared`) or a shared library (with `-shared`). Its use requires more rigorous testing of shared library dependencies. In addition to testing shared libraries for matching versions, timestamps and checksums must also match the timestamps and checksums recorded in shared library dependency records at link time.

You can use the `odump` command to examine a shared library's versions string, as set by using the `-set_version` *`version-string`* option of the `ld` command that created the library. For example:

```
% odump -D library-name
```

The value displayed for the `IVERSION` field is the version string specified when the library was built. If a shared library is built without the `-set_version` option, no `IVERSION` field will be displayed. These shared libraries are handled as if they had been built with the version identifier `_null`.

When `ld` links a shared object, it records the version of each shared library dependency. Only the rightmost version identifier in a colon-separated list is recorded. To examine these dependencies for any shared executable file or library, use the following command:

```
% odump -Dl shared-object-name
```

4.11.3 Major and Minor Versions Identifiers

Tru64 UNIX does not distinguish between major and minor versions of shared libraries:

- Major versions are used to distinguish incompatible versions of shared libraries.
- Minor versions typically distinguish different but compatible versions of a library. Minor versions are often used to provide revision-specific identification or to restrict the use of backward-compatible shared libraries.

Tru64 UNIX shared libraries use a colon-separated list of version identifiers to provide the versioning features normally attained through minor versions.

The following sequence of library revisions shows how revision-specific identification can be added to the version list of a shared library without affecting shared library compatibility:

Shared Library	Version
`libminor.so`	3.0
`libminor.so`	3.1:3.0
`libminor.so`	3.2:3.1:3.0

Each new release of `libminor.so` adds a new identifier at the beginning of the version list. The new identifier distinguishes the latest revision from its predecessors. Any executable files linked against any revision of `libminor.so` will record `3.0` as the required version, so no distinction is made between the compatible libraries. The additional version identifiers are only informational.

The following sequence of library revisions shows how the use of backward-compatible shared libraries can be restricted:

Shared Library	Version
`libminor2.so`	3.0
`libminor2.so`	3.0:3.1
`libminor2.so`	3.0:3.1:3.2

In this example, programs linked with old versions of libminor2.so can be executed with newer versions of the library, but programs linked with newer versions of libminor2.so cannot be executed with any of the previous versions.

4.11.4 Full and Partial Versions of Shared Libraries

You can implement a binary compatible version of a shared library as a complete, independent object or as a partial object that depends directly or indirectly on a complete, independent object. A fully duplicated shared library takes up more disk space than a partial one, but involves simpler dependency processing and uses less swap space. The reduced disk space requirements are the only advantage of a partial version of a shared library.

A partial shared library includes the minimum subset of modules required to provide backward compatibility for applications linked prior to a binary incompatible change in a newer version of the library. It is linked against one or more earlier versions of the same library that provide the full set of library modules. By this method, you can chain together multiple versions of shared libraries so that any instance of the shared library will indirectly provide the full complement of symbols normally exported by the library.

For example, version osf.1 of libxyz.so includes modules x.o, y.o, and z.o. It was built and installed using the following commands:

```
% ld -shared -o libxyz.so -set_version osf.1 \
    x.o y.o z.o -lc
% mv libxyz.so /usr/shlib/libxyz.so
```

If, at some future date, libxyz.so requires an incompatible change that affects only module z.o, a new version, called osf.2, and a partial version, still called osf.1, can be built as follows:

```
% ld -shared -o libxyz.so -set_version osf.2 x.o \
    y.o new_z.o -lc
% mv libxyz.so /usr/shlib/libxyz.so
% ld -shared -o libxyz.so -set_version osf.1 z.o \
    -lxyz -lc
% mv libxyz.so /usr/shlib/osf.1/libxyz.so
```

4.11.5 Linking with Multiple Versions of Shared Libraries

In general, applications are linked with the newest versions of shared libraries. Occasionally, you might need to link an application or shared library with an older, binary compatible version of a shared library. In such a case, use the ld command's -L option to identify older versions of the shared libraries used by the application.

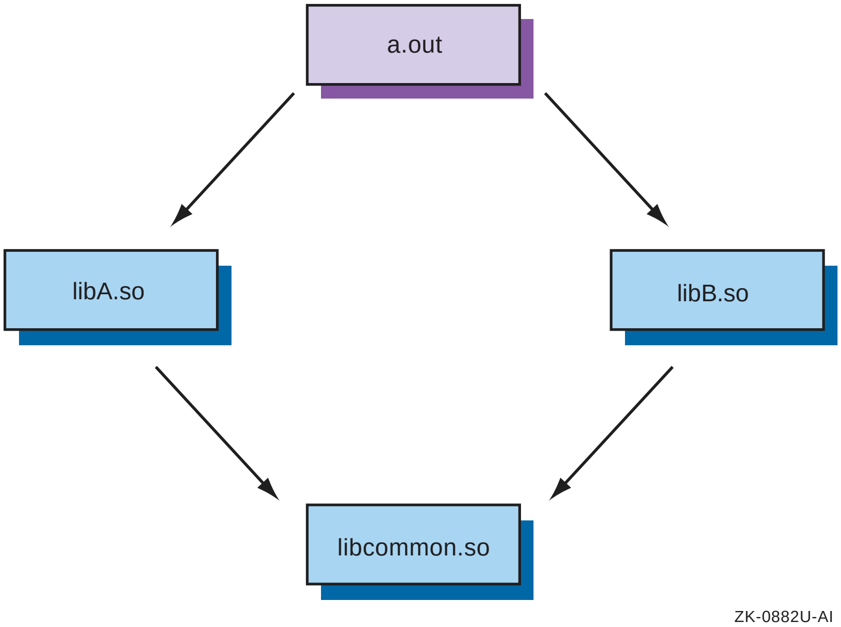
The linker issues a warning when you link an application with more than one version of the same shared library. In some cases, the multiple version dependencies of an application or shared library will not be noticed until it is loaded for execution.

By default, the `ld` command tests for multiple version dependencies only for those libraries it is instructed to link against. To identify all possible multiple version dependencies, use the `ld` command's `-transitive_link` option to include indirect shared library dependencies in the link step.

When an application is linked with partial shared libraries, the linker must carefully distinguish dependencies on multiple versions resulting from partial shared library implementations. The linker reports multiple version warnings when it cannot differentiate between acceptable and unacceptable multiple version dependencies.

In some instances, multiple version dependencies might be reported at link time for applications that do not use multiple versions of shared libraries at run time. Consider the libraries and dependencies shown in Figure 4–2 and described in the following table.

Figure 4–2: Linking with Multiple Versions of Shared Libraries



Library	Version	Dependency	Dependent Version
libA.so	v1	libcommon.so	v1
libB.so	v2	libcommon.so	v2
libcommon.so	v1:v2	—	—

In the preceding table, libA.so was linked against a version of libcommon.so that had a rightmost version identifier of v1. Unlike libA.so, libB.so was linked against a version of libcommon.so that had a rightmost version identifier of v2. Because the libcommon.so shown in the table includes both v1 and v2 in its version string, the dependencies of both libA.so and libB.so are satisfied by the one instance of libcommon.so.

When a.out is linked, only libA.so and libB.so are mentioned on the link command line. However, the linker examines the dependencies of libA.so and libB.so, recognizes the possible multiple version dependency on libcommon.so, and issues a warning. By linking a.out against libcommon.so as well, you can avoid this false warning.

4.11.6 Version Checking at Load Time

The loader performs version matching between the list of versions supported by a shared library and the versions recorded in shared library dependency records. If a shared object is linked with the -exact_match option on the link command line, the loader also compares the timestamp and checksum of a shared library against the timestamp and checksum values saved in the dependency record.

After mapping in a shared library that fails the version-matching test, the loader attempts to locate the correct version of the shared library by continuing to search other directories in RPATH, LD_LIBRARY_PATH, or the default search path.

If all of these directories are searched without finding a matching version, the loader attempts to locate a matching version by appending the version string recorded in the dependency to the directory path at which the first nonmatching version of the library was located.

For example, a shared library libfoo.so is loaded in directory /usr/local/lib with version osf.2, but a dependency on this library requires version osf.1. The loader attempts to locate the correct version of the library using a constructed path like the following:

```
/usr/local/lib/osf.1/libfoo.so
```

If this constructed path fails to locate the correct library or if no version of the library is located at any of the default or user-specified search directories, the loader makes one last attempt to locate the library by appending the required version string to the standard system shared library directory (/usr/shlib). This last attempt will therefore use a constructed path like the following:

```
/usr/shlib/osf.1/libfoo.so
```

If the loader fails to find a matching version of a shared library, it aborts the load and reports a detailed error message indicating the dependency and shared library version that could not be located.

You can disable version checking for programs that are not installed with the setuid function by setting the loader environment variable as shown in the following C shell example:

```
% setenv _RLD_ARGS -ignore_all_versions
```

You can also disable version checking for specific shared libraries as shown in the following example:

```
% setenv _RLD_ARGS -ignore_version libDXm.so
```

4.11.7 Multiple Version Checking at Load Time

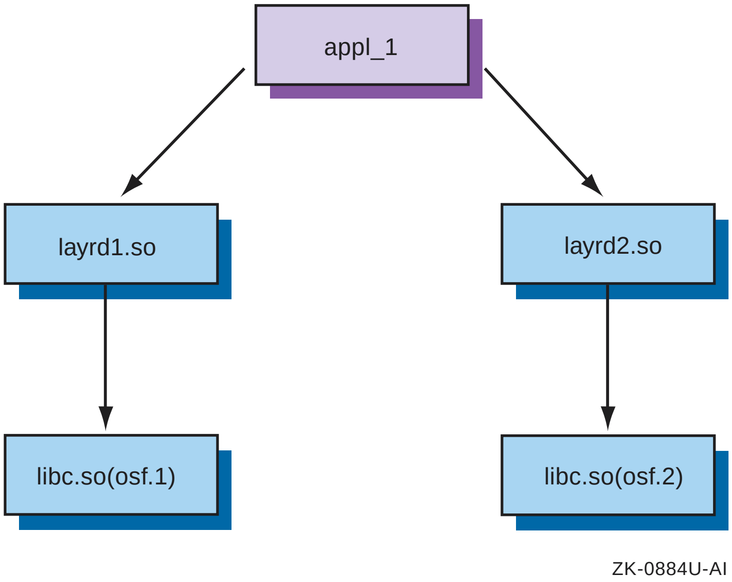
Like the linker, the loader must distinguish between valid and invalid uses of multiple versions of shared libraries:

- Valid uses of multiple versions occur when partial shared libraries that depend on other versions of the same libraries are loaded. In some cases, these partial shared libraries depend on different partial shared libraries, and the result can be complicated dependency relationships that the loader must interpret carefully to avoid reporting false errors.
- Invalid uses of multiple versions occur when two different shared objects depend on different versions of another shared object. Partial shared library chains are an exception to this rule. For version-checking purposes, the first partial shared library in a chain defines a set of dependencies that overide similar dependencies in other members of the chain.

The following figures show shared object dependencies that will result in multiple dependency errors. Version identifiers are shown in parentheses.

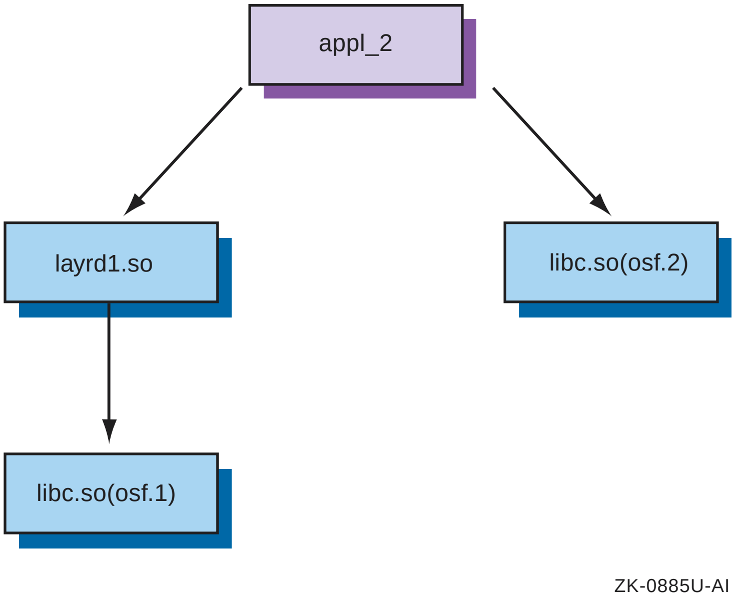
In Figure 4–3, an application uses two layered products that are built with incompatible versions of the base system.

Figure 4–3: Invalid Multiple Version Dependencies Among Shared Objects: Example 1



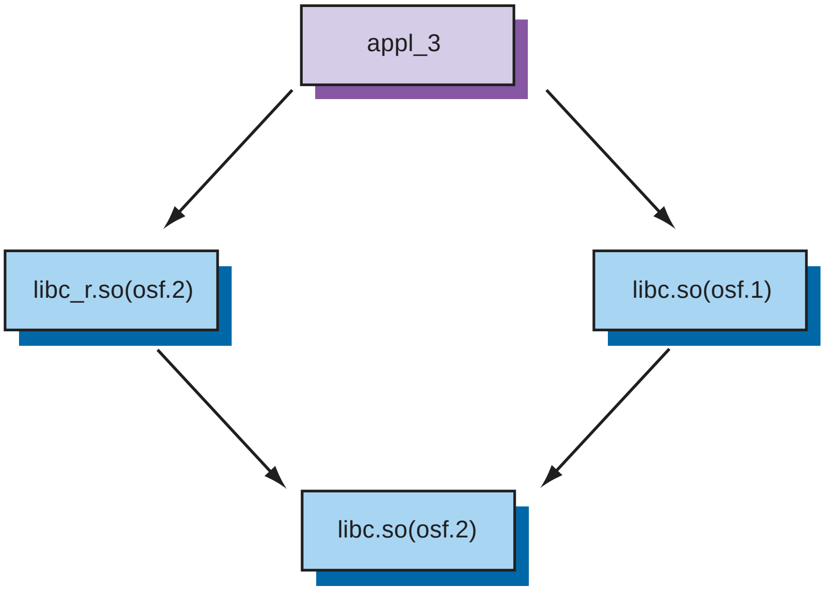
In Figure 4–4, an application is linked with a layered product that was built with an incompatible version of the base system.

Figure 4–4: Invalid Multiple Version Dependencies Among Shared Objects: Example 2



In Figure 4–5, an application is linked with an incomplete set of backward-compatible libraries that are implemented as partial shared libraries.

Figure 4–5: Invalid Multiple Version Dependencies Among Shared Objects: Example 3

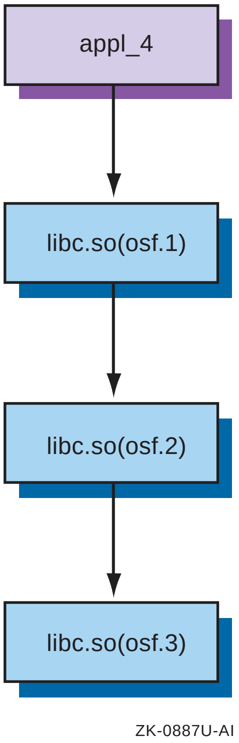


ZK-0886U-AI

The following figures show valid uses of multiple versions of shared libraries.

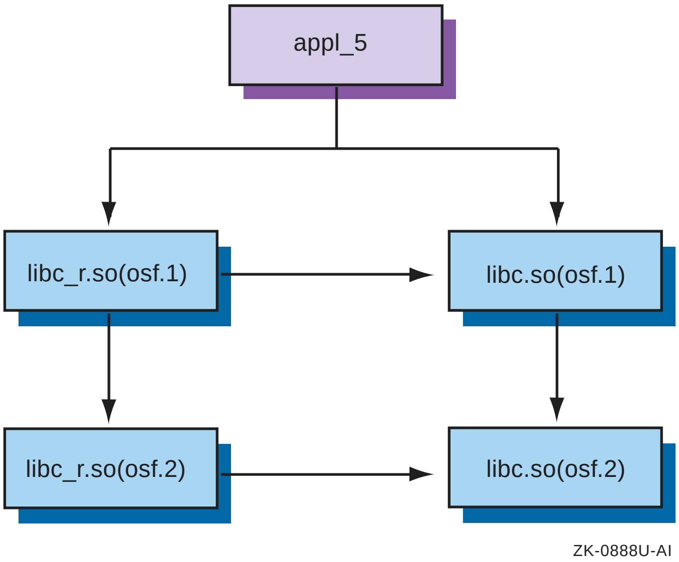
In Figure 4–6, an application uses a backward-compatibility library implemented as a partial shared library.

Figure 4–6: Valid Uses of Multiple Versions of Shared Libraries: Example 1



In Figure 4–7, an application uses two backward-compatible libraries, one of which depends on the other.

Figure 4–7: Valid Uses of Multiple Versions of Shared Libraries: Example 2



4.12 Symbol Binding

The loader can resolve symbols using either deferred or immediate binding. Immediate binding requires that all symbols be resolved when an executable program or shared library is loaded. Deferred (lazy) binding allows text symbols to be resolved at run time. A lazy text symbol is resolved the first time that a reference is made to it in a program.

By default, programs are loaded with deferred binding. Setting the `LD_BIND_NOW` environment variable to a non-null value selects immediate binding for subsequent program invocations.

Immediate binding can be useful to identify unresolvable symbols. With deferred binding in effect, unresolvable symbols might not be detected until a particular code path is executed.

Immediate binding can also reduce symbol-resolution overhead. Run-time symbol resolution is more expensive per symbol than load-time symbol resolution.

4.13 Shared Library Restrictions

The use of shared libraries is subject to the following restrictions:

- Shared libraries should not have any undefined symbols.

  Shared libraries should be explicitly linked with other shared libraries that define the symbols they refer to.

In certain cases, such as a shared library that refers to symbols in an executable file, it is difficult to avoid references to undefined symbols. See Section 4.2.4 for a discussion on how to handle unresolved external symbols in a shared library.

- Certain files (such as assembler files, older object files, and C files) that were optimized at level `O3` might not work with shared libraries.

  C modules compiled with the Tru64 UNIX C compiler at optimization level `O2` or less will work with shared libraries. Executable programs linked with shared libraries can be compiled at optimization level `O3` or less.

- Programs that are installed using the `setuid` or `setgid` subroutines do not use the settings of the various environment variables that govern library searches (such as `LD_LIBRARY_PATH`, `_RLD_ARGS`, `_RLD_LIST`, and `_RLD_ROOT`); they use only system-installed libraries (that is, those in `/usr/shlib`). This restriction prevents potential threats to the security of these programs, and it is enforced by the run-time loader (`/sbin/loader`).

5

Debugging Programs with dbx

The dbx debugger is a command-line program. It is a tool for debugging programs at the source-code level and machine-code level, and can be used with C, Fortran, Pascal, and assembly language. After invoking dbx, you can enter dbx commands that control and trace execution, display variable and expression values, and display and edit source files.

The ladebug debugger, an alternate debugger, provides both command-line and graphical user interfaces (GUIs) and supports some languages that are not supported by dbx. The ladebug debugger has better features than dbx for debugging multithreaded programs. For more information about ladebug, see the *Ladebug Debugger Manual* or ladebug(1).

This chapter provides information on the following topics:

- General debugging considerations (Section 5.1)
- How to run the dbx debugger (Section 5.2)
- What you can specify in dbx commands (Section 5.3)
- How to enter dbx commands using options provided by the dbx monitor (Section 5.4)
- How to control dbx (Section 5.5)
- How to examine source code and machine code (Section 5.6)
- How to control the execution of the program you are debugging (Section 5.7)
- How to set breakpoints (Section 5.8)
- How to examine the state of a program (Section 5.9)
- How to preserve multiple core files (Section 5.10)
- How to debug a running process (Section 5.11)
- How to debug multithreaded processes (Section 5.12)
- How to debug multiple asynchronous processes (Section 5.13)
- The sample C program (sam.c) that is referred to in examples throughout this chapter (Section 5.14)

Complete details on dbx command-line options, dbx commands, variables, and so on can be found in dbx(1).

You can also use Visual Threads (available on the Associated Products Volume 1 CDROM) to analyze multithreaded applications for potential logic and performance problems. You can use Visual Threads with POSIX Threads Library applications and with Java applications.

Examples in this chapter refer to a sample program called `sam`. The C language source program (`sam.c`) is listed in Example 5–1.

In addition to the conventions outlined in the preface of this manual, an additional convention is used in the command descriptions in this chapter; uppercase keywords are used to indicate variables for which specific rules apply. These keywords are described in Table 5–1.

Table 5–1: Keywords Used in Command Syntax Descriptions

Keyword	Value
ADDRESS	Any expression specifying a machine address.
COMMAND_LIST	One or more commands, each separated by semicolons.
DIR	Directory name.
EXP	Any expression including program variable names for the command. Expressions can contain `dbx` variables, for example, (`$listwindow + 2`). If you want to use the variable names `in`, `to`, or `at` in an expression, you must surround them with parentheses; otherwise, `dbx` assumes that these words are debugger keywords.
FILE	File name.
INT	Integer value.
LINE	Source-code line number.
NAME	Name of a `dbx` command.
PROCEDURE	Procedure name or an activation level on the stack.
REGEXP	Regular expression string. See `ed`(1).
SIGNAL	System signal. See `signal`(2).
STRING	Any ASCII string.
VAR	Valid program variable or `dbx` predefined variable (see Table 5-9). For machine-level debugging, VAR can also be an address. You must qualify program variables with duplicate names as described in Section 5.3.2.

The following example shows the use of the uppercase words in commands:

```
(dbx) stop VAR in PROCEDURE if EXP
```

Enter `stop`, `in`, and `if` as shown. Enter the values for VAR, PROCEDURE, and EXP as defined in Table 5–1.

Note

Information on debugging multiple asynchronous processes, including extensions to the syntax of certain dbx commands to provide control of the asynchronous session, is contained in Section 5.13.

5.1 General Debugging Considerations

The following sections introduce the dbx debugger and some debugging concepts. They also give suggestions about how to approach a debugging session, including where to start, how to isolate errors, and how to avoid common pitfalls. If you are an experienced programmer, you may not need to read these sections.

5.1.1 Reasons for Using a Source-Level Debugger

The dbx debugger enables you to trace problems in a program object at the source-code level or at the machine-code level. With dbx, you control a program's execution, monitoring program control flow, variables, and memory locations. You can also use dbx to trace the logic and flow of control to become familiar with a program written by someone else.

5.1.2 Explanation of Activation Levels

Activation levels define the currently active scopes (usually procedures) on the stack. An activation stack is a list of calls that starts with the initial program, usually main(). The most recently called procedure or block is number 0. The next procedure called is number 1. The last activation level is always the main procedure (the procedure that controls the whole program). Activation levels can also consist of blocks that define local variables within procedures. You see activation levels in stack traces (see the where and tstack debugger commands), and when moving around the activation stack (see the up, down, and func debugger commands). The following example shows a stack trace produced by a where command:

```
>  0 prnt(pline = 0x11ffffcb8) ["sam.c":52, 0x120000c04]   1
   1 main(argc = 2, argv = 0x11ffffe08) ["sam.c":45, 0x120000bac]2
   |  |             |                     |      |   |
   3  4             5                     6      7   8
```

1 The most recently called procedure is prnt. The activation level of prnt is 0; this function is at the top of the stack.

2 The main program is main.

3 Activation level number. The angle bracket (>) indicates the activation level that is currently under examination.

4 Procedure name.

5 Procedure arguments.

6 Source file name.

7 Current line number.

8 Current program counter.

5.1.3 Isolating Program Execution Failures

Because the `dbx` debugger finds only run-time errors, you should fix compiler errors before starting a debugging session. Run-time errors can cause a program to fail during execution (resulting in the creation of a core dump file) or to produce incorrect results. The approach for debugging a program that fails during execution differs from the approach for debugging a program that executes to completion but produces incorrect results. (See Section 5.1.4 for information on how to debug programs that produce incorrect results.)

If a program fails during execution, you can usually save time by using the following approach to start a debugging session instead of blindly debugging line by line:

1. Invoke the program under `dbx`, specifying any appropriate options and the names of the executable file and the core dump file on the `dbx` command line.
2. Get a stack trace using the `where` command to locate the point of failure.

> **Note**
>
> If you have not stripped symbol table information from the object file, you can get a stack trace even if the program was not compiled with the `–g` debug option.

3. Set breakpoints to isolate the error using the `stop` or `stopi` commands.
4. Display the values of variables using the `print` command to see where a variable might have been assigned an incorrect value.

If you still cannot find the error, other `dbx` commands described in this chapter might be useful.

5.1.4 Diagnosing Incorrect Output Results

If a program executes to completion but produces incorrect values or output, follow these steps:

1. Set a breakpoint where you think the problem is happening — for example, in the code that generates the value or output.
2. Run the program.
3. Get a stack trace using the `where` command.
4. Display the values for the variables that might be causing the problem using the `print` command.
5. Repeat this procedure until the problem is found.

5.1.5 Creating a Core Snapshot of a Running Process

In some cases, a program that is not failing or producing incorrect results might not be behaving properly. For example:

- The program might be executing very slowly.
- A server might have stopped responding.

Or, the program might even be hung in an infinite loop.

In such cases you might want to examine the state of the program without stopping its execution. The `coredump` command lets you create a core snapshot of a running process. By default, the core snapshot file is created in the current directory and is named `corefile`. You can then use `dbx` to examine the core snapshot as explained in Section 5.2.3. For more information, see `coredump`(1).

5.1.6 Avoiding Pitfalls

The debugger cannot solve all problems. For example, if your program contains logic errors, the debugger can only help you find the problem, not solve it. When information displayed by the debugger appears confusing or incorrect, taking the following actions might correct the situation:

- Separate lines of source code into logical units wherever possible (for example, after `if` conditions). The debugger may not recognize a source statement written with several others on the same line.
- If executable code appears to be missing, it might have been contained in an included file. The debugger treats an included file as a single line of code. If you want to debug this code, remove it from the included file and compile it as part of the program.

- Make sure you recompile the source code after changing it. If you do not do this, the source code displayed by the debugger will not match the executable code. The debugger warns you if the source file is more recent than the executable file.
- If you stop the debugger by pressing Ctrl/Z and then resume the same debugging session, the debugger continues with the same object module specified at the start of the session. This means that if you stop the debugger to fix a problem in the code, recompile, and resume the session, the debugger will not reflect the change. You must start a new session.

  Similarly, dbx will not reflect changes you have made if you edit and recompile your program in one window on a workstation while running the debugger in another window. You must stop and restart dbx each time you want it to recognize changes you have made.
- When entering a command to display an expression that has the same name as a dbx keyword, you must enclose the expression within parentheses. For example, to display the value of output (a keyword in the playback and record commands, discussed in Section 5.9.4), you must specify the following command:

  ```
  (dbx) print (output)
  ```
- If the debugger does not display any variables or executable code, make sure you compiled the program with the –g option.

5.2 Running dbx

Before invoking dbx, you need to compile the program for debugging. You might also want to create a dbx initialization file that will execute commands when the debugger is started.

5.2.1 Compiling a Program for Debugging

To prepare a program for debugging, specify the –g option at compilation time. With this option set, the compiler inserts into the program symbol table information that the debugger uses to locate variables. With the –g option set, the compiler also sets its optimization level to –O0. When you use different levels of optimizing, for example –O2, the optimizer does not alter the flow of control within a program, but it might move operations around so that the object code and source code do not correspond. These changed sequences of code can create confusion when you use the debugger.

You can do limited debugging on code compiled without the –g option. For example, the following commands work properly without recompiling for debugging:

- `stop in PROCEDURE`

- `stepi`
- `cont`
- `conti`
- `(ADDRESS)/<COUNT><MODE>`
- `tracei`

Although you can do limited debugging, it is usually more advantageous to recompile the program with the `–g` option. Note that the debugger does not warn you if an object file was compiled without the `–g` option.

Complete symbol table information is available only for programs in which all modules have been compiled with the `–g` option. Other programs will have symbol table information only for symbols that are either referenced by or defined in modules compiled with the `–g` option.

Note

Any routines in shared library applications in which breakpoints are to be set must be compiled with the `–g` option. If the `–g` option is not specified, the symbol table information that `dbx` needs to set breakpoints is not generated and `dbx` will not be able to stop the application.

5.2.2 Creating a dbx Initialization File

You can create a `dbx` initialization file that contains commands you normally enter at the beginning of each `dbx` session. For example, the file could contain the following commands:

```
set $page = 5
set $lines = 20
set $prompt = "DBX> "
alias du dump
```

The initialization file must have the name `.dbxinit`. Each time you invoke the debugger, `dbx` executes the commands in `.dbxinit`. The debugger looks first for `.dbxinit` in the current directory and then in your home directory (the directory assigned to the `$HOME` environment variable).

5.2.3 Invoking and Terminating dbx

You invoke `dbx` from the shell command line by entering the `dbx` command and any necessary parameters.

After invocation, `dbx` sets the current function to the first procedure of the program.

The dbx command has the following syntax:

dbx [*options*] [*objectfile* [*corefile*]]

options
: Several of the most important options supported by the dbx command line are shown in Table 5–2.

objectfile
: Specifies the executable file of the program that you want to debug. If *objectfile* is not specified, dbx uses a.out by default.

corefile
: Specifies a core dump file, created if a program failed during execution or if the coredump(1) command was used to take a core snapshot of a running process. The dump file holds an image of memory at the time the program failed or the snapshot was taken. If you specify a core file on the command line, dbx identifies where the program failed or the snapshot was taken, and you can then use dbx commands to determine the state of the program at that time. Core dump files are all named core by default. See Section 5.10 for information about enabling core-file naming at the system or application level.

The maximum number of arguments accepted by dbx is 1000; however, system limits on your machine might reduce this number.

Table 5–2: dbx Command Options

Option	Function
-c*filename*	Selects an initialization command file other than your .dbxinit file.
-I*dirname*	Tells dbx to look in the specified directory for source files. To specify multiple directories, use a separate −I for each directory. Unless you specify this option when you invoke dbx, the debugger looks for source files in the current directory and in the object file's directory. You can change directories with the use command (see Section 5.6.1).
−i	Invokes dbx in interactive mode. With this option set, dbx does not treat source lines beginning with number signs (#) as comments.
−k	Maps memory addresses. This option is useful for kernel debugging. (For information on kernel debugging, see the *Kernel Debugging* manual.)

Table 5–2: dbx Command Options (cont.)

Option	Function
–module_path	Specifies the directory path where dbx should search for shared libraries (or loadable kernel modules). This option is useful if, for example, you are debugging a core dump (or a kernel crash dump) and the version of a shared library (or module) that was running when the dump occurred has been moved to a different location. (For information on kernel debugging, see the *Kernel Debugging* manual.)
–module_verbose	Causes dbx to print the path of shared libraries (or loadable kernel modules, if you are debugging a kernel) as they are being loaded. By default, dbx does not print the path. (For information on kernel debugging, see the *Kernel Debugging* manual.)
-pid *process-id*	Attaches dbx to a currently running process.
–r	Immediately executes the object file that you specify on the command line. If program execution terminates with an error, dbx displays the message that describes the error. You can then either invoke the debugger or allow the program to continue exiting. The dbx debugger reads from /dev/tty when you specify the –r option and standard input is not a terminal. If the program executes successfully, dbx prompts you for input.

The following example invokes dbx with no options. Because an object file name is not specified, dbx prompts for one. In this case, the user responds with sam. The default debugger prompt is (dbx).

```
% dbx
enter object file name (default is ’a.out’): sam
dbx version 3.12
Type ’help’ for help.

main:  23  if (argc < 2) {
(dbx)
```

Use the quit or q command to end a debugging session. The quit command accepts no arguments.

5.3 Using dbx Commands

You can enter up to 10,240 characters on an input line. Long lines can be continued with a backslash (\). If a line exceeds 10,240 characters, dbx displays an error message. The maximum string length is also 10,240.

The following sections describe scoping and the use of qualified variable names, dbx expressions and precedence, and dbx data types and constants.

5.3.1 Qualifying Variable Names

Variables in dbx are qualified by file, procedure, block, or structure. When using commands like print to display a variable's value, dbx indicates the scope of the variable when the scope could be ambiguous (for example, you have a variable by the same name in two or more procedures). If the scope is wrong, you can specify the full scope of the variable by separating scopes with periods. For example:

```
sam.main.i
 |    |    |
 1    2    3
```

1 Current file

2 Procedure name

3 Variable name

5.3.2 dbx Expressions and Their Precedence

The dbx debugger recognizes expression operators from C; these operators can also be used for debugging any other supported language. (Note that dbx uses brackets ([ ]) for array subscripts even in Fortran, whose natural subscript delimiters are parentheses.) In addition to the standard C operators, dbx uses the number sign (#) as shown in Table 5–3.

Table 5–3: The dbx Number-Sign Expression Operator

Syntax	Description
("FILE" #EXP)	Uses the line number specified by #EXP in the file named by FILE.
(PROCEDURE #EXP)	Uses the relative line number specified by #EXP in the procedure named by PROCEDURE.
(#EXP)	Returns the address for the line specified by (#EXP).

Operators follow the C language precedence. Table 5–4 shows the language operators recognized by dbx in order of precedence from top to bottom and from left to right, with the dbx-specific number-sign operator included among the unary operators to show its place in the precedence hierarchy.

Table 5–4: Expression Operator Precedence

Unary:	&, +, –, * (pointer), #, `sizeof()`[a], ~, /, `(type)`, `(type *)`
Binary:	<<, >>, ", !, ==, !=, <=, >=, <, >, &, &&, |, ||, +, –, *,/[b], %, [], –>

[a] The `sizeof` operator specifies the number of bytes retrieved to get an element, not `(`*`number-of-bits`*`+7)/8`.

[b] For backward compatibility, `dbx` also accepts two slashes (`//`) as a division operator.

5.3.3 dbx Data Types and Constants

Table 5–5 lists the built-in data types that `dbx` commands can use.

Table 5–5: Built-in Data Types

Data Type	Description	Data Type	Description
`$address`	Pointer	`$real`	Double-precision real
`$boolean`	Boolean	`$short`	16-bit integer
`$char`	Character	`$signed`	Signed integer
`$double`	Double-precision real	`$uchar`	Unsigned character
`$float`	Single-precision real	`$unsigned`	Unsigned integer
`$integer`	Signed integer	`$void`	Empty

You can use the built-in data types for type coercion — for example, to display the value of a variable in a type other than the type specified in the variable's declaration. The `dbx` debugger understands C language data types, so that you can refer to data types without the dollar sign (`$`). The types of constants that are acceptable as input to `dbx` are shown in Table 5–6. Constants are displayed by default as decimal values in `dbx` output.

Table 5–6: Input Constants

Constant	Description
`false`	0
`true`	Nonzero
`nil`	0
`0x`*`number`*	Hexadecimal
`0t`*`number`*	Decimal
`0`*`number`*	Octal
`number`	Decimal
`number``.[`*`number`*`][e|E][+|-]EXP`	Float

Notes:

- Overflow on nonfloat uses the rightmost digits. Overflow on float uses the leftmost digits of the mantissa and the highest or lowest exponent possible.
- The `$octin` variable changes the default input expected to octal. The `$hexin` variable changes the default input expected to hexadecimal (see Section 5.5.2).
- The `$octints` variable changes the default output to octal. The `$hexints` variable changes the default output to hexadecimal (see Section 5.5.2).

5.4 Working with the dbx Monitor

The `dbx` debugger provides a command history, command-line editing, and symbol name completion. The `dbx` debugger also allows multiple commands on an input line. These features can reduce the amount of input required or allow you to repeat previously executed commands.

5.4.1 Repeating dbx Commands

The `dbx` debugger keeps a command history that allows you to repeat debugger commands without retyping them. You can display these commands by using the `history` command. The `$lines` variable controls the number of history lines saved. The default is 20 commands. You can use the `set` command to modify the `$lines` variable (see Section 5.5.1).

To repeat a command, use the Return key or one of the exclamation point (`!`) commands.

The `history` command has the following forms:

`history`
Displays the commands in the history list.

Return key
Repeats the last command that you entered. You can disable this feature by setting the `$repeatmode` variable to 0 (see Section 5.5.1).

`!`*`string`*
Repeats the most recent command that starts with the specified string.

`!`*`integer`*
Repeats the command associated with the specified integer.

`!–`*`integer`* Repeats the command that occurred the specified number of commands (*`integer`*) before the most recent command.

The following example displays the history list and then repeats execution of the twelfth command in the list:

```
(dbx) history
  10 print x
  11 print y
  12 print z
(dbx) !12
(!12 = print z)
123
(dbx)
```

5.4.2 Editing the dbx Command Line

The `dbx` debugger provides support for command-line editing. You can edit a command line to correct mistakes without reentering the entire command. To enable command-line editing, set the `EDITOR`, `EDITMODE`, or `LINEEDIT` environment variable before you invoke `dbx`. For example, to set `LINEEDIT` from the C shell, enter the following command:

```
% setenv LINEEDIT
```

From the Bourne or Korn shells, enter this command:

```
$ export LINEEDIT
```

The debugger offers the following modes of command-line editing:

- If the environment variable `LINEEDIT` is not set and either of the environment variables `EDITMODE` or `EDITOR` contains a path ending in `vi`, the debugger uses a command-line editing mode that resembles the Korn shell's `vi` mode, in which the following editing keys are recognized:

  ```
  $ + - 0 A B C D E F I R S W X ^
  a b c d e f h i j k l r s w x ~
  Ctrl/D
  Ctrl/H
  Ctrl/J
  Ctrl/L
  Ctrl/M
  Ctrl/V
  ```

  See `ksh`(1) for more information.

- If the environment variable `LINEEDIT` is set to any value, even the null string, or if `LINEEDIT` is not set and either of the environment variables `EDITMODE` or `EDITOR` contains a path ending in `emacs`, the debugger uses a command-line editing mode that resembles the Korn shell's `emacs` mode. This mode behaves slightly differently depending on whether it is enabled by `LINEEDIT` or by `EDITOR` or `EDITMODE`.

Table 5–7 lists the `emacs`-mode command-line editing commands.

Table 5–7: Command-Line Editing Commands in emacs Mode

Command	Function
Ctrl/A	Moves the cursor to the beginning of the command line.
Ctrl/B	Moves the cursor back one character.
Ctrl/C	Clears the line.
Ctrl/D	Deletes the character at the cursor.
Ctrl/E	Moves the cursor to the end of the line.
Ctrl/F	Moves the cursor ahead one character.
Ctrl/H	Deletes the character immediately preceding the cursor.
Ctrl/J	Executes the line.
Ctrl/K	(When enabled by `EDITOR` or `EDITMODE`) Deletes from the cursor to the end of the line. If preceded by a numerical parameter whose value is less than the current cursor position, deletes from the given position up to the cursor. If preceded by a numerical parameter whose value is greater than the current cursor position, deletes from the cursor up to the given position.
Ctrl/K *`char`*	(When enabled by `LINEEDIT`) Deletes characters until the cursor rests on the next occurrence of *`char`*.
Ctrl/L	Redisplays the current line.
Ctrl/M	Executes the line.
Ctrl/N	Moves to the next line in the history list.
Ctrl/P	Moves to the previous line in the history list.
Ctrl/R *`char`*	Searches back in the current line for the specified character.
Ctrl/T	Interchanges the two characters immediately preceding the cursor.
Ctrl/U	Repeats the next character four times.
Ctrl/W	Deletes the entire line.
Ctrl/Y	Inserts immediately before the cursor any text cut with Ctrl/K.
Ctrl/Z	Tries to complete a file or symbol name.
Escape	Tries to complete a file or symbol name.
Down Arrow	Moves to the next line in the history list.
Up Arrow	Moves to the previous line in the history list.

Table 5–7: Command-Line Editing Commands in emacs Mode (cont.)

Command	Function
Left Arrow	Moves the cursor back one character.
Right Arrow	Moves the cursor ahead one character.

5.4.3 Entering Multiple Commands

You can enter multiple commands on the command line by using a semicolon (;) as a separator. This feature is useful when you are using the `when` command (see Section 5.8.4).

The following example has two commands on one command line; the first command stops the program and the second command reruns it:

```
(dbx) stop at 40; rerun
[2] stop at "sam.c":40
[2] stopped at   [main:40 ,0x120000b40]  i=strlen(line1.string);
(dbx)
```

5.4.4 Completing Symbol Names

The `dbx` debugger provides symbol name completion. When you enter a partial symbol name and press Ctrl/Z, `dbx` attempts to complete the name. If a unique completion is found, `dbx` redisplays the input with the unique completion added; otherwise, all possible completions are shown, and you can choose one.

To enable symbol name completion, you must enable command-line editing as described in Section 5.4.2. The following example displays all names beginning with the letter `i`:

```
(dbx) i  Ctrl/Z
ioctl.ioctl .ioctl isatty.isatty .isatty i int 1
(dbx) i                                        2
```

1 The display might include data types and library symbols.

2 After listing all names beginning with the partial name, `dbx` prompts again with the previously specified string, giving you an opportunity to specify additional characters and repeat the search.

The following example shows symbol name completion. In this case, the entry supplied is unambiguous:

```
(dbx) print file  Ctrl/Z
(dbx) print file_header_ptr
0x124ac
(dbx)
```

5.5 Controlling dbx

The `dbx` debugger provides commands for setting and removing `dbx` variables, creating and removing aliases, invoking a subshell, checking and deleting items from the status list, displaying a list of object files associated with an application, and recording and playing back input.

5.5.1 Setting and Removing Variables

The `set` command defines a `dbx` variable, sets an existing `dbx` variable to a different value, or displays a list of existing `dbx` predefined variables. The `unset` command removes a `dbx` variable. Use the `print` command to display the values of program and debugger variables. The `dbx` predefined variables are listed in Table 5–8. You cannot define a debugger variable with the same name as a program variable.

The `set` and `unset` commands have the following forms:

`set`
Displays a list of `dbx` predefined variables.

`set VAR = EXP`
Assigns a new value to a variable or defines a new variable.

`unset VAR`
Unsets the value of a `dbx` variable.

The following example shows the use of the `set` and `unset` commands:

```
(dbx) set               1
$listwindow      10
$datacache        1
$main        "main"
$pagewindow      22
test              5
$page             1
$maxstrlen      128
$cursrcline      24
more (n if no)? n
(dbx) set test = 12     2
(dbx) set
$listwindow      10
$datacache        1
$main        "main"
$pagewindow      22
test             12
$page             1
$maxstrlen      128
$cursrcline      24
more (n if no)? n
```

```
(dbx) unset test       3
(dbx) set
$listwindow      10
$datacache        1
$main        "main"
$pagewindow      22
$page             1
$maxstrlen      128
$cursrcline      24
more (n if no)? n
(dbx)
```

1 Display a list of dbx predefined variables.

2 Assign a new value to a variable.

3 Remove a variable.

5.5.2 Predefined dbx Variables

The predefined dbx variables are shown in Table 5–8. Each variable is labeled I for integer, B for Boolean, or S for string. Variables that you can examine but cannot modify are indicated by an R.

Table 5–8: Predefined dbx Variables

Type	Name	Default	Description
S	`$addrfmt`	`"0x%lx"`	Specifies the format for addresses. Can be set to anything you can format with a C language `printf` statement.
B	`$assignverify`	1	Specifies whether new values are displayed when assigning a value to a variable.
B	`$asynch_interface`	0	Controls whether dbx is, or can be, configured to control multiple asynchronous processes. Incremented by 1 when a process is attached; decremented by 1 when a process terminates or is detached. Can also be set by the user. If 0 or negative, asynchronous debugging is disabled.

Table 5–8: Predefined dbx Variables (cont.)

Type	Name	Default	Description
B	`$break_during_step`	0	Controls whether breakpoints are checked while processing `step/stepi`, `next/nexti`, `call`, `return`, and so on.
B	`$casesense`	0	Specifies whether source searching and variables are case sensitive. A nonzero value means case sensitive; a 0 means not case sensitive.
I R	`$curevent`	0	Shows the last event number as reported by the `status` command.
I R	`$curline`	0	Shows the current line in the source code.
I R	`$curpc`	–	Shows the current address. Used with the `wi` and `li` aliases.
I R	`$cursrcline`	1	Shows the last line listed plus 1.
B	`$datacache`	1	Caches information from the data space so that `dbx` only has to check the data space once. If you are debugging the operating system, set this variable to 0; otherwise, set it to a nonzero value.
S R	`$defaultin`	Null string	Shows the name of the file that `dbx` uses to store information when using the `record input` command.
S R	`$defaultout`	Null string	Shows the name of the file that `dbx` uses to store information when using the `record output` command.
B	`$dispix`	0	When set to 1, specifies the display of only real instructions when debugging in `pixie` mode.

Table 5–8: Predefined dbx Variables (cont.)

Type	Name	Default	Description
B	`$hexchars`	Not defined	A nonzero value indicates that character values are shown in hexadecimal.
B	`$hexin`	Not defined	A nonzero value indicates that input constants are hexadecimal.
B	`$hexints`	Not defined	A nonzero value indicates that output constants are shown in hexadecimal; a nonzero value overrides octal.
B	`$hexstrings`	Not defined	A nonzero value indicates that strings are displayed in hexadecimal; otherwise, strings are shown as characters.
I R	`$historyevent`	None	Shows the current history number.
I	`$lines`	20	Specifies the size of the `dbx` history list.
I	`$listwindow`	`$pagewindow/2`	Specifies the number of lines shown by the `list` command.
S	`$main`	`"main"`	Specifies the name of the procedure where execution begins. The debugger starts the program at `main()` unless otherwise specified.
I	`$maxstrlen`	128	Specifies the maximum number of characters that `dbx` prints for pointers to strings.

Table 5–8: Predefined dbx Variables (cont.)

Type	Name	Default	Description
S	`$module_path`	Null string	Specifies the directory path where `dbx` should search for shared libraries (or loadable kernel modules). This variable is useful if, for example, you are debugging a core dump (or a kernel crash dump) and the version of a shared library (or module) that was running when the dump occurred has been moved to a different location. For information on kernel debugging, see the *Kernel Debugging* manual.
I	`$module_verbose`	0	When set to a nonzero value, causes `dbx` to print the location of shared libraries (or loadable kernel modules, if debugging a kernel) as they are being loaded. By default, or when this variable is set to 0, `dbx` does not print the location. For information on kernel debugging, see the *Kernel Debugging* manual.
B	`$octin`	Not defined	Changes the default input constants to octal when set to a nonzero value. Hexadecimal overrides octal.
B	`$octints`	Not defined	Changes the default output constants to octal when set to a nonzero value. Hexadecimal overrides octal.
B	`$page`	1	Specifies whether to page long information. A nonzero value enables paging; a 0 disables it.

Table 5–8: Predefined dbx Variables (cont.)

Type	Name	Default	Description
I	`$pagewindow`	Various	Specifies the number of lines displayed when viewing information that is longer than one screen. This variable should be set to the number of lines on the terminal. A value of 0 indicates a minimum of 1 line. The default value depends on the terminal type; for a standard video display, the default is 24.
B	`$pimode`	0	Displays input when using the `playback input` command.
I	`$printdata`	0	A nonzero value indicates that the values of registers are displayed when instructions are disassembled; otherwise, register values are not displayed.
B	`$printtargets`	1	If set to 1, specifies that displayed disassembly listings are to include the labels of targets for jump instructions. If set to 0, disables this label display.
B	`$printwhilestep`	0	For use with the `step` [*n*] and `stepi` [*n*] instructions. A nonzero value specifies that all *n* lines or instructions should be displayed. A 0 value specifies that only the last line and/or instruction should be displayed.
B	`$printwide`	0	Specifies wide (useful for structures or arrays) or vertical format for displaying variables. A nonzero value indicates wide format; 0 indicates vertical format.
S	`$prompt`	`"(dbx)"`	Sets the prompt for `dbx`.

Table 5–8: Predefined dbx Variables (cont.)

Type	Name	Default	Description
B	`$readtextfile`	1	When set to a value of 1, `dbx` tries to read instructions from the object file instead of from the process. This variable should always be set to 0 when the process being debugged copies in code during the debugging process. However, performance is better when `$readtextfile` is set to 1.
B	`$regstyle`	1	Specifies the type of register names to be used. A value of 1 specifies hardware names. A 0 specifies software names as defined by the file `regdefs.h`.
B	`$repeatmode`	1	Specifies whether `dbx` should repeat the last command when the Return key is pressed. A nonzero value indicates that the command is repeated; otherwise, it is not repeated.
B	`$rimode`	0	Records input when using the `record output` command.
S	`$sigvec`	`"sigaction"`	Tells `dbx` the name of the code called by the system to set signal handlers.
S	`$sigtramp`	`"_sigtramp"`	Tells `dbx` the name of the code called by the system to invoke user signal handlers.

Table 5–8: Predefined dbx Variables (cont.)

Type	Name	Default	Description
B	`$stopall_on_step`	1	Specifies whether `dbx` should stop every child process that is forked (1) or ignore many of the forks generated by various system and library calls (0). If `$stop_all_forks` is not set, the value of `$stop_on_fork` determines `dbx`'s behavior with forks. `$stop_all_forks` traps forks in libraries and system calls that are usually ignored by `$stop_on_fork`.
B	`$stop_in_main`	N/A	Not used. This variable is displayed by the `set` command, but it presently has no effect on `dbx` operation.
B	`$stop_on_exec`	1	Specifies whether `dbx` should detect calls to `execl( )` and `execv( )`, and stop the newly activated images at the first line of executable code.
B	`$stop_on_fork`	1	Specifies whether `dbx` should advance a new image activated by a `fork( )` or `vfork( )` call to its main activation point and then stop (1) or continue until stopped by a breakpoint or event (0). The `dbx` program tries to avoid stopping on forks from system or library calls unless `$stop_all_forks` is set.

Table 5–8: Predefined dbx Variables (cont.)

Type	Name	Default	Description
S	`$tagfile`	`"tags"`	Contains a file name indicating the file in which the `tag` command and the `tagvalue` macro are to search for tags.
I	`$traploops`	3	Specifies the number of consecutive calls to a `SIGTRAP` handler that will be made before `dbx` assumes that the program has fallen into a trap-handling loop.

5.5.3 Defining and Removing Aliases

The `alias` command defines a new alias or displays a list of all current aliases.

The `alias` command allows you to rename any debugger command. Enclose commands containing spaces within double- or single-quotation marks. You can also define a macro as part of an alias.

The `dbx` debugger has a group of predefined aliases. You can modify these aliases or add new aliases. You can also include aliases in your `.dbxinit` file for use in future debugging sessions. The `unalias` command removes an alias from a command. You must specify the alias to remove. The alias is removed only for the current debugging session.

The `alias` and `unalias` commands have the following forms:

`alias`

Displays a list of all aliases.

`alias NAME1[(ARG1,...,ARGN)] "NAME2"`

Defines a new alias. `NAME1` is the new name. `NAME2` is the command to string to rename. `ARG1,...,ARGN` are the command arguments.

`unalias NAME`

Removes an alias from a command, where `NAME` is the alias name.

The following example shows the use of the `alias` and `unalias` commands:

```
(dbx) alias                     1
h       history
```

```
si      stepi
Si      nexti
⋮

g       goto
s       step
More (n if no) ?n
(dbx) alias ok(x) "stop at x"      2
(dbx) ok(52)                       3
[2] Stop at "sam.c":52             4
(dbx)
(dbx) unalias h                    5
(dbx) alias
si      stepi
Si      nexti
⋮

g       goto
s       step
More (n if no)? n
(dbx)
```

1. Display aliases.
2. Define an alias for setting a breakpoint.
3. Set a breakpoint at line 52.
4. Debugger acknowledges breakpoint set at line 52.
5. Remove the `h` alias. (Note that it disappears from the alias list.)

5.5.4 Monitoring Debugging Session Status

The `status` command checks which, if any, of the following commands are currently set:

- `stop` or `stopi` commands for breakpoints
- `trace` or `tracei` commands for line-by-line variable tracing
- `when` command
- `record input` and `record output` commands for saving information in a file

The `status` command accepts no arguments. For example:

```
(dbx) status
[2] trace i in main
[3] stop in prnt
[4] record output /tmp/dbxt0018898 (0 lines)
(dbx)
```

The numbers in brackets (for example, [2]) indicate status item numbers.

5.5.5 Deleting and Disabling Breakpoints

The delete command deletes breakpoints and stops the recording of input and output. Deleting a breakpoint or stopping recording removes the pertinent items from the status list produced by the status command.

The disable command disables breakpoints without deleting them. The enable command reenables disabled events.

The delete command has the following forms:

```
delete EXP1[,...,EXPN]
```

Deletes the specified status items.

```
delete all
delete *
```

Deletes all status items.

The following example shows the use of the delete command:

```
(dbx) status
[2] record output /tmp/dbxt0018898 (0 lines)
[3] trace i in main
[4] print pline at "sam.c":
[5] stop in prnt
(dbx) delete 4
(dbx) status
[2] record output /tmp/dbxt0018898 (0 lines)
[3] trace i in main
[5] stop in prnt
(dbx)
```

The disable and enable commands have the following forms:

```
disable EVENT1[,EVENT2,...]
enable EVENT1[,EVENT2,...]
```

Disables or enables the specified events.

```
disable all
enable all
```

Disables or enables all events.

5.5.6 Displaying the Names of Loaded Object Files

The listobj command displays the names of all object files that have been loaded by dbx, together with their sizes and the address at which they were loaded. These objects include the main program and all of the shared libraries that are used in an application. The listobj command accepts no arguments. For example:

```
(dbx) listobj
sam                          addr: 0x120000000      size: 0x2000
/usr/shlib/libc.so           addr: 0x3ff80080000    size: 0xbc000
(dbx)
```

5.5.7 Specifying the Location of Shared Libraries for Core Dumps

When a core dump occurs, the location of any shared libraries used by the program is recorded in the core file, enabling dbx to find the libraries. If the version of a shared library that was running when the dump occurred is moved to a different location, dbx will not find it. You can specify the directory path where dbx should look for shared libraries by using any one of the following methods (see dbx(1) for complete details):

- On the dbx command line, specify the directory path with the -module_path option. For example:

  ```
   % dbx a.out core -module_path /usr/project4/lib_dir
  ```

- Before invoking dbx, set the environment variable DBX_MODULE_PATH. For example:

  ```
  % setenv DBX_MODULE_PATH /usr/project4/lib_dir
  ```

- During the dbx session, if you want to load a shared library dynamically, first set the $module_path dbx variable and then use the addobj command to load the library, as in the following example:

  ```
  (dbx) set $module_path /usr/project4/lib_dir
  (dbx) addobj libdef.so
  ```

To verify that modules are being loaded from the correct location, turn on verbose module-loading using any one of the following methods:

- Specify the -module_verbose dbx command option.
- Set the DBX_MODULE_VERBOSE environment variable to any integer value.
- Set the $module_verbose dbx variable to a nonzero value.

5.5.8 Invoking a Subshell from Within dbx

To invoke an interactive subshell at the dbx prompt, enter sh. To return to dbx from a subshell, enter exit or press Ctrl/D. To invoke a subshell that performs a single command and returns to dbx, enter sh and the desired shell command. For example:

```
(dbx) sh
% date
```

```
Tue Aug 9 17:25:15 EDT 1998
% exit:
(dbx) sh date
Tue Aug 9 17:29:34 EDT 1998
(dbx)
```

5.6 Examining Source Programs

The following sections describe how to list and edit source code, change directories, change source files, search for strings in source code, display qualified symbol names, and display type declarations.

5.6.1 Specifying the Locations of Source Files

If you did not specify the −I option when invoking dbx (see Section 5.2.3), the debugger looks for source files in the current directory or the object file's directory. The use command has two functions:

- Change the directory or list of directories in which the debugger looks
- List the directory or directories currently in use

The command recognizes absolute and relative pathnames (for example, ./), but it does not recognize the C shell tilde (~).

The use command has the following forms:

`use`

Lists the current directories.

`use DIR1 ...  DIRN`

Replaces the current list of directories with a new set.

For example:

```
(dbx) use
.                            1
(dbx) use /usr/local/lib
(dbx) use
/usr/local/lib               2
(dbx)
```

1 Current directory

2 New directory

5.6.2 Moving Up or Down in the Activation Stack

As described in Section 5.1.2, the debugger maintains a stack of activation levels. To find the name or activation number for a specific procedure, get a stack trace with the `where` or `tstack` command. You can move through the activation stack by using the `up`, `down`, and `func` commands.

5.6.2.1 Using the where and tstack Commands

The `where` command displays a stack trace showing the current activation levels (active procedures) of the program being debugged. The `tstack` command displays a stack trace for all threads. See Section 5.12 for more information about debugging threads.

The `where` and `tstack` commands have the following form:

`where [EXP]`

`tstack [EXP]` Displays a stack trace.

If `EXP` is specified, `dbx` displays only the top `EXP` levels of the stack; otherwise, the entire stack is displayed.

If a breakpoint is set in `prnt` in the sample program `sam.c`, the program runs and stops in the procedure `prnt()`. If you enter `where`, the debugger's stack trace provides the information shown in the following example:

```
(dbx) stop in prnt
[1] stop in prnt
⋮

(dbx) run

(dbx) where 1
```

```
>  0 prnt(pline = 0x11ffffcb8) ["sam.c":52, 0x120000c04]
   |  |            |             |      |   |
  [1][2]          [3]           [4]    [5] [6]
(dbx)
```

1 Activation level

2 Procedure name

3 Current value of the argument `pline`

4 Source file name

5 Line number

6 Program counter

5.6.2.2 Using the up, down, and func Commands

The `up` and `down` commands move you directly up or down in the stack; they are useful when tracking a call from one level to another.

The `func` command can move you up or down incrementally or to a specific activation level or procedure. The `func` command changes the current line, the current file, and the current procedure, which changes the scope of the variables you can access. You can also use the `func` command to examine source code when a program is not executing.

The `up`, `down`, and `func` commands have the following forms:

`up [EXP]` Moves up the specified number of activation levels in the stack. The default is one level.

`down [EXP]` Moves down the specified number of activation levels in the stack. The default is one level.

`func` Displays the current activation levels.

`func PROCEDURE` Moves to the activation level specified by `PROCEDURE`.

`func EXP` Moves to the activation level specified by the expression.

The following example shows the use of these commands:

```
(dbx) where
>  0 prnt(pline = 0x11ffffcb8) ["sam.c":52, 0x120000c04]
   1 main(argc = 2, argv = 0x11ffffe08) ["sam.c":45, 0x120000bac]
(dbx) up
```

```
main:  45  prnt(&line1);                                  1
(dbx) where
   0 prnt(pline = 0x11ffffcb8) ["sam.c":52, 0x120000c04]
>  1 main(argc = 2, argv = 0x11ffffe08) ["sam.c":45, 0x120000bac]
(dbx) down
prnt:  52  fprintf(stdout,"%3d.  (%3d) %s",               2
(dbx) where
>  0 prnt(pline = 0x11ffffcb8) ["sam.c":52, 0x120000c04]
   1 main(argc = 2, argv = 0x11ffffe08) ["sam.c":45, 0x120000bac]
(dbx) func 1
main 47        prnt(&line1)                               3
(dbx)
```

1 Move up one level.

2 Move down one level.

3 Move directly to `main`.

5.6.3 Changing the Current Source File

The `file` command displays the current source file name or changes the current source file.

Note

> Before setting a breakpoint or trace on a line number, use the `func` command to get the correct procedure. The `file` command cannot be specific enough for the debugger to access the information necessary to set a breakpoint.

The `file` command has the following forms:

`file`	Displays the name of the file currently in use.
`file FILE`	Changes the current file to the specified file.

For example:

```
(dbx) file
sam.c              1
(dbx) file data.c
(dbx) file
data.c             2
(dbx)
```

1 Current file

2 New file

5.6.4 Listing Source Code

The `list` command displays lines of source code. The `dbx` variable `$listwindow` defines the number of lines that `dbx` lists by default. The `list` command uses the current file, procedure, and line, unless otherwise specified.

The `list` command has the following forms:

`list` Lists the number of lines specified by `$listwindow`, starting at the current line.

`list EXP` Lists the number of lines specified by `EXP`, starting at the current line.

`list EXP1,EXP2` List lines from `EXP1` to `EXP2`.

`list EXP:INT` Starting at the specified line (`EXP`), lists the specified number of lines (`INT`), overriding `$listwindow`.

`list PROCEDURE` Lists the specified procedure for `$listwindow` lines.

The following example specifies a two-line list starting at line 49:

```
(dbx) list 49:2
    49  void prnt(pline)
    50  LINETYPE *pline;
```

If you use the `list` command's predefined alias `w`, the output is as follows:

```
(dbx) w
    45    prnt(&line1);
    46   }
    47  }
    48
    49  void prnt(pline)
>   50  LINETYPE *pline;
    51  {
 *  52   fprintf(stdout,"%3d.  (%3d) %s",pline->linenumber,
    53       pline->length, pline->string);
    54   fflush(stdout);
```

The right angle bracket in column 1 (`>`) indicates the current line, and the asterisk in column 2 (`*`) indicates the location of the program counter (PC) at this activation level.

5.6.5 Searching for Text in Source Files

The slash (/) and question mark (?) commands search for regular expressions in source code. The slash searches forward from the current line, and the question mark searches backward. Both commands wrap around at the end of the file if necessary, searching the entire file from the point of invocation back to the same point. By default, dbx does not distinguish uppercase letters from lowercase when searching. If you set the dbx variable $casesense to any nonzero value, the search is case sensitive.

The / and ? commands have the following form:

/[REGEXP] Searches forward for the specified regular expression or, if no expression is specified, for the regular expression associated with the last previous search command.

?[REGEXP] Searches backward in the same manner as the slash command's forward search.

For example:

```
(dbx) /lines
no match
(dbx) /line1
  16  LINETYPE line1;
(dbx) /
  39  while(fgets(line1.string, sizeof(line1.string), fd) != NULL){
(dbx)
```

5.6.6 Editing Source Files from Within dbx

The edit command enables you to change source files from within dbx. To make the changes effective, you must quit from dbx, recompile the program, and restart dbx.

The edit command has the following forms:

edit Invokes an editor on the current file.

edit FILE Invokes an editor on the specified file.

The edit command loads the editor indicated by the environment variable EDITOR or, if EDITOR is not set, the vi editor. To return to dbx, exit normally from the editor.

5.6.7 Identifying Variables That Share the Same Name

The `which` and `whereis` commands display program variables. These commands are useful for debugging programs that have multiple variables with the same name occurring in different scopes. The commands follow the rules described in Section 5.3.1.

The `which` and `whereis` commands have the following forms:

`which VAR` Displays the default version of the specified variable.

`whereis VAR` Displays all versions of the specified variable.

In the following example, the user checks to see where the default variable named `i` is and then verifies that this is the only instance of `i` in the program by observing that `whereis` shows only the one occurrence:

```
(dbx) which i
sam.main.i
(dbx) whereis i
sam.main.i
```

5.6.8 Examining Variable and Procedure Types

The `whatis` command lists the type declaration for variables and procedures in a program.

The `whatis` command has the following form:

`whatis VAR` Displays the type declaration for the specified variable or procedure.

For example:

```
(dbx) whatis main
int main(argc,argv)
int argc;
unsigned char **argv;
(dbx) whatis i
int i;
(dbx)
```

5.7 Controlling the Program

The following sections describe the `dbx` commands used to run a program, step through source code, return from a procedure call, start at a specified line, continue after stopping at a breakpoint, assign values to program variables, patch an executable disk file, execute a particular routine, set an environment variable, and load shared libraries.

5.7.1 Running and Rerunning the Program

The `run` and `rerun` commands start program execution. Each command accepts program arguments and passes those arguments to the program. If no arguments are specified for a `run` command, `dbx` runs the program with no arguments. If no arguments are specified for a `rerun` command, `dbx` defaults to the arguments used with the previous `run` or `rerun` command. You can specify arguments in advance of entering a `rerun` command by using the `args` command. Arguments set by the `args` command are ignored by a subsequent `run` command.

You can also use these commands to redirect program input and output in a manner similar to redirection in the C shell:

- The optional parameter `<FILE1` redirects input to the program from the specified file.
- The optional parameter `>FILE2` redirects output from the program to the specified file.
- The optional parameter `>&FILE2` redirects both `stderr` and `stdout` to the specified file.

Note

The redirected output differs from the output saved with the `record output` command (see Section 5.9.4.2), which saves debugger output, not program output.

The `run`, `args`, and `rerun` commands have the following forms:

```
run [ARG1 ...  ARGN] [<FILE1] [>FILE2]
run [ARG1 ...  ARGN] [<FILE1] [>&FILE2]
```

Runs the program with the specified arguments and redirections.

```
args [ARG1 ...  ARGN] [<FILE1] [>FILE2]
args [ARG1 ...  ARGN] [<FILE1] [>&FILE2]
```

Sets the specified arguments and redirections for use by subsequent commands; the specified values remain in effect until explicitly altered by new values given with a `run` or `rerun` command.

```
rerun [ARG1 ...  ARGN] [<FILE1] [>FILE2]
rerun [ARG1 ...  ARGN] [<FILE1] [>&FILE2]
```

Reruns the program with the specified arguments and redirections.

For example:

```
(dbx) run sam.c               1
0.  (19)#include <stdio.h>
1.  (14) struct line {
2.  (19) char string[256];
⋮

Program terminated normally
(dbx) rerun                   2
0.  (19)#include <stdio.h>
1.  (14) struct line {
2.  (19) char string[256];
⋮

Program terminated normally
(dbx)
```

1 The argument is `sam.c`.

2 Reruns the program with the previously specified arguments.

5.7.2 Executing the Program Step by Step

For debugging programs written in high-level languages, the `step` and `next` commands execute a fixed number of source-code lines as specified by EXP. For debugging programs written in assembly language, the `stepi` and `nexti` commands work the same as `step` and `next` except that they step by machine instructions instead of by program lines. If EXP is not specified, `dbx` executes one source-code line or machine instruction; otherwise, `dbx` executes the source-code lines or machine instructions as follows:

- The `dbx` debugger does not take comment lines into consideration in interpreting EXP. The program executes EXP source-code lines, regardless of the number of comment lines interspersed among them.
- For `step` and `stepi`, `dbx` considers EXP to apply both to the current procedure and to called procedures. Program execution stops after EXP source lines in the current procedure and any called procedures.
- For `next` and `nexti`, `dbx` considers EXP to apply only to the current procedure. Program execution stops after executing EXP source lines in the current procedure, regardless of the number of source lines executed in any called procedures.

The `step/stepi` and `next/nexti` commands have the following form:

`step [EXP]`

`stepi [EXP]` Executes the specified number of lines or instructions in both the current procedure and any called procedures. The default is 1.

`next [EXP]`

`nexti [EXP]`
Executes the specified number of source-code lines or machine instructions in only the current procedure, regardless of the number of lines executed in any called procedures. The default is 1.

For example:

```
(dbx) rerun
[7] stopped at [prnt:52,0x120000c04] fprintf(stdout,"%3d.(%3d) %s",
(dbx) step 2
  0.  ( 19) #include <stdio.h>
  [prnt:55 ,0x120000c48]  }
(dbx) step
  [main:40 ,0x120000b40]        i=strlen(line1.string);
(dbx)
```

The `$break_during_step` and `$printwhilestep` variables affect stepping. See Table 5–8 for more information.

5.7.3 Using the return Command

The `return` command is used in a called procedure to execute the remaining instructions in the procedure and return to the calling procedure.

The `return` command has the following forms:

`return`
Executes the rest of the current procedure and stops at the next sequential line in the calling procedure.

`return PROCEDURE`
Executes the rest of the current procedure and any calling procedures intervening between the current procedure and the procedure named by `PROCEDURE`. Stops at the point of the call in the procedure that is named.

For example:

```
(dbx) rerun
[7] stopped at [prnt:52,0x120000c04] fprintf(stdout,"%3d.(%3d) %s",
(dbx) return
  0.  (19) #include <stdio.h>
stopped at   [main:45 +0xc,0x120000bb0]  prnt(&line1);
(dbx)
```

5.7.4 Going to a Specific Place in the Code

The `goto` command shifts to the specified line and continues execution. This command is useful in a `when` statement — for example, to skip a line known to cause problems. The `goto` command has the following form:

`goto LINE` Goes to the specified source line when you continue execution.

For example:

```
(dbx) when at 40 {goto 43}
[8] start sam.c:43 at "sam.c":40
(dbx)
```

5.7.5 Resuming Execution After a Breakpoint

For debugging programs written in high-level languages, the `cont` command resumes program execution after a breakpoint. For debugging programs written in assembly language, the `conti` command works the same as `cont`. The `cont` and `conti` commands have the following forms:

```
cont
conti
```

Continues from the current source-code line or machine-code address.

```
cont to LINE
conti to ADDRESS
```

Continues until the specified source-code line or machine-code address.

```
cont in PROCEDURE
conti in PROCEDURE
```

Continues until the specified procedure.

```
cont SIGNAL
conti SIGNAL
```

After receiving the specified signal, continues from the current line or machine instruction.

```
cont SIGNAL to LINE
conti SIGNAL to ADDRESS
```

After receiving the specified signal, continues until the specified line or address.

```
cont SIGNAL in PROCEDURE
conti SIGNAL in PROCEDURE
```

Continues until the specified procedure and sends the specified signal.

The following example shows the use of the `cont` command in a C program:

```
(dbx) stop in prnt
[9] stop in prnt
(dbx) rerun
[9] stopped at [prnt:52,0x120000c04] fprintf(stdout,"%3d.(%3d) %s",
(dbx) cont
  0.  ( 19) #include <stdio.h>
[9] stopped at [prnt:52,0x120000c04] fprintf(stdout,"%3d.(%3d) %s",
(dbx)
```

The following example shows the use of the `conti` command in an assembly-language program:

```
(dbx) conti
  0.  ( 19) #include <stdio.h>
[4] stopped at >*[prnt:52 ,0x120000c04]      ldq   r16,-32640(gp)
(dbx)
```

5.7.6 Changing the Values of Program Variables

The `assign` command changes the value of a program variable. The `assign` command has the following form:

```
assign VAR = EXP
assign EXP1 = EXP2
```

Assigns a new value to the program variable named by `VAR` or the address represented by the resolution of `EXP1`.

For example:

```
(dbx) print i
19                                 1
(dbx) assign i = 10
10                                 2
(dbx) assign *(int *)0x444 = 1     3
 1
(dbx)
```

1 The value of `i`.

2 The new value of `i`.

3 Coerce the address to be an integer and assign a value of 1 to it.

5.7.7 Patching Executable Disk Files

The `patch` command patches an executable disk file to correct bad data or instructions. Only text, initialized data, or read-only data areas can be patched. The `bss` segment cannot be patched because it does not exist in disk files. The `patch` command fails if it is entered against a program that is executing.

The `patch` command has the following form:

```
patch VAR = EXP
patch EXP1 = EXP2
```

Assigns a new value to the program variable named by `VAR` or the address represented by the resolution of `EXP1`.

The patch is applied to the default disk file; you can use qualified variable names to specify a patch to a file other than the default. Applying a patch in this way also patches the in-memory image of the file being patched.

For example:

```
(dbx) patch &main = 0
(dbx) patch var = 20
(dbx) patch &var = 20
(dbx) patch 0xnnnnn = 0xnnnnn
```

5.7.8 Running a Specific Procedure

It is possible for you to set the current line pointer to the beginning of a procedure, place a breakpoint at the end of the procedure, and run the procedure. However, it is usually easier to use the `call` or `print` command to execute a procedure in your program. The `call` or `print` command executes the procedure you specify on the command line. You can pass parameters to the procedure by specifying them as arguments to the `call` or `print` command.

The `call` or `print` command does not alter the flow of your program. When the procedure returns, the program remains stopped at the point where you entered the `call` or `print` command. The `print` command displays values returned by called procedures; the `call` command does not.

The `call` and `print` commands have the following forms:

```
call PROCEDURE([parameters ])
print PROCEDURE([parameters ])
```

Executes the object code associated with the named procedure or function. Specified parameters are passed to the procedure or function.

For example:

```
(dbx) stop in prnt                         1
[11] stop in prnt
(dbx) call prnt(&line1)                    2
[11] stopped at [prnt:52,0x120000c] fprintf(stdout,"%3d.(%3d) %s",
(dbx) status                               3
[11] stop in prnt
[12] stop at "sam.c":40
[2] record output example2 (126 lines)
(dbx) delete 11,12                         4
(dbx)
```

1 The `stop` command sets a breakpoint in the `prnt()` function.

2 The `call` command begins executing the object code associated with `prnt()`. The `line1` argument passes a string by reference to `prnt`.

3 The `status` command displays the currently active breakpoints.

4 The `delete` command deletes the breakpoints at lines 52 and 40.

The `print` command allows you to include a procedure as part of an expression to be printed. For example:

```
(dbx) print sqrt(2.)+sqrt(3.)
```

5.7.9 Setting Environment Variables

Use the `setenv` command to set an environment variable. You can use this command to set the value of an existing environment variable or create a new environment variable. The environment variable is visible to both `dbx` and the program you are running under `dbx` control, but it is not visible after you exit the `dbx` environment. However, if you start a shell with the `sh` command within `dbx`, that shell can see `dbx` environment variables. To change an environment variable for a process, you must enter the `setenv` command before starting up the process within `dbx` with the `run` command.

The `setenv` command has the following form:

`setenv VAR "STRING"`

> Changes the value of an existing environment variable or creates a new one. To reset an environment variable, specify a null string.

For example:

```
(dbx) setenv TEXT "sam.c"                   1
(dbx) run                                   2
[4] stopped at [prnt:52,0x120000e34] fprintf(stdout,"%3d.(%3d) %s",
(dbx) setenv TEXT ""                        3
(dbx) run                                   4
Usage: sam filename
```

```
Program exited with code 1
```

1 The setenv command sets the environment variable TEXT to the value sam.c.

2 The run command executes the program from the beginning. The program reads input from the file named in the the environment variable TEXT. Program execution stops at the breakpoint at line 52.

3 The setenv command sets the environment variable TEXT to null.

4 The run command executes the program. Because the TEXT environment variable contains a null value, the program must get input.

5.8 Setting Breakpoints

A breakpoint stops program execution and lets you examine the program's state at that point. The following sections describe the dbx commands to set a breakpoint at a specific line or in a procedure and to stop for signals.

5.8.1 Overview

When a program stops at a breakpoint, the debugger displays an informational message. For example, if a breakpoint is set in the sample program sam.c at line 23 in the main() procedure, the following message is displayed:

```
[4] stopped at   [main:40, 0x120000b18]     i=strlen(line1.string);
 |                 |   |    |               |
 1                 2   3    4               5
```

1 Breakpoint status number.

2 Procedure name.

3 Line number.

4 Current program counter. Use this number to display the assembly-language instructions from this point. (See Section 5.7.5 for more information.)

5 Source line.

Before setting a breakpoint in a program with multiple source files, be sure that you are setting the breakpoint in the right file. To select the right procedure, take the following steps:

1. Use the file command to select the source file.
2. Use the func command to specify a procedure name.
3. List the lines of the file or procedure using the list command (see Section 5.6.4).

4. Use a `stop at` command to set a breakpoint at the desired line.

5.8.2 Setting Breakpoints with stop and stopi

For debugging programs written in high-level languages, the `stop` command sets breakpoints to stop execution as follows: at a source line, in a procedure, when a variable changes, or when a specified condition is true. For debugging programs written in assembly language, the `stopi` command works the same as `stop`, except that it traces by machine instructions instead of by program lines. You can also instruct `dbx` to stop when it enters a new image invoked by an `exec( \)` call by setting the `$stop_on_exec` predefined variable (see Table 5–8).

- The `stop at` and `stopi at` commands set a breakpoint at a specific source-code line or machine-code address, as applicable. The `dbx` debugger stops only at lines or addresses that have executable code. If you specify a nonexecutable stopping point, `dbx` sets the breakpoint at the next executable point. If you specify the `VAR` parameter, the debugger displays the variable and stops only when `VAR` changes; if you specify `if EXP`, the debugger stops only when `EXP` is true.
- The `stop in` and `stopi in` commands set a breakpoint at the beginning or, conditionally, for the duration of a procedure.
- The `stop if` and `stopi if` commands cause `dbx` to stop program execution under specified conditions. Because `dbx` must check the condition after the execution of each line, this command slows program execution markedly. Whenever possible, use `stop/stopi at` or `stop/stopi in` instead of `stop/stopi if`.
- If the `$stop_on_exec` predefined variable is set to 1, an `exec( )` call causes `dbx` to stop and read in the new image's symbol table, then advance to the image's main activation point and stop for user input.

The `delete` command removes breakpoints established by the `stop` or `stopi` command.

The `stop` and `stopi` commands have the following forms:

```
stop VAR
stopi VAR
```

Stops when `VAR` changes.

```
stop VAR at LINE
stopi VAR at ADDRESS
```

Stops when `VAR` changes at a specified source-code line or machine-code address.

```
stop VAR at LINE if EXP
stopi VAR at ADDRESS if EXP
```

Stops when VAR changes at a specified line or address only if the expression is true.

```
stop if EXP
stopi if EXP
```

Stops if EXP is true.

```
stop VAR if EXP
stopi VAR if EXP
```

Stops when VAR changes if EXP is true.

```
stop in PROCEDURE
stopi in PROCEDURE
```

Stops at the beginning of the procedure.

```
stop VAR in PROCEDURE
```

Stops in the specified procedure when VAR changes.

```
stop VAR in PROCEDURE if EXP
stopi VAR in PROCEDURE if EXP
```

Stops when VAR changes in the specified procedure if EXP is true.

Note

Specifying both VAR and EXP causes stops anywhere in the procedure, not just at the beginning. Using this feature is time consuming because the debugger must check the condition before and after each source line is executed. (When both arguments are specified, EXP is always checked before VAR.)

The following example shows the use of `stop` in a C program:

```
(dbx) stop at 52
[3] stop at "sam.c":52
(dbx) rerun
[3] stopped at [prnt:52,0x120000fb0] fprintf(stdout,"%3d.(%3d) %s",
(dbx) stop in prnt
[15] stop in prnt
(dbx)
```

The following example shows the use of `stopi` in an assembly-language program:

```
(dbx) stopi at 0x120000c04
[4] stop at 0x120000c04
(dbx) rerun
[7] stopped at >*[prnt:52 ,0x120000c04]   ldq    r16, -32640(gp)
```

5.8.3 Tracing Variables During Execution

For debugging programs written in high-level languages, the `trace` command lists the value of a variable while the program is executing and determines the scope of the variable being traced. For debugging programs written in assembly language, the `tracei` command works the same as `trace`, except that it traces by machine instructions instead of by program lines.

The `trace` and `tracei` commands have the following forms:

`trace LINE`

Lists the specified source line each time it is executed.

`trace VAR`
`tracei VAR`

Lists the specified variable after each source line or machine instruction is executed.

`trace [VAR] at LINE`
`tracei [VAR] at ADDRESS`

Lists the specified variable at the specified line or instruction.

`trace [VAR] in PROCEDURE`
`tracei [VAR] in PROCEDURE`

Lists the specified variable in the specified procedure.

`trace [VAR] at LINE if EXP`
`tracei [VAR] at ADDRESS if EXP`

Lists the variable at the specified source-code line or machine-code address when the expression is true and the value of the variable has changed. (EXP is checked before VAR.)

`trace [VAR] in PROCEDURE if EXP`
`tracei [VAR] in PROCEDURE if EXP`

Lists the variable in the specified procedure when the expression is true and the value of the variable has changed. (EXP is checked before VAR.)

For example:

```
(dbx) trace i
[5] trace i in main
(dbx) rerun sam.c
[4]   [main:25 ,0x400a50]
(dbx) c
[5] i changed before [main: line 41]:
  new value = 19;
[5] i changed before [main: line 41]:
  old value = 19;
  new value = 14;
[5] i changed before [main: line 41]:
  old value = 14;
  new value = 19;
[5] i changed before [main: line 41]:
  old value = 19;
  new value = 13;
[5] i changed before [main: line 41]:
  old value = 13;
  new value = 17;
[5] i changed before [main: line 41]:
  old value = 17;
  new value = 3;
[5] i changed before [main: line 41]:
  old value = 3;
  new value = 1;
[5] i changed before [main: line 41]:
  old value = 1;
  new value = 30;
```

5.8.4 Writing Conditional Code in dbx

The `when` command controls the conditions under which certain `dbx` commands that you specify will be executed.

The `when` command has the following forms:

`when VAR [if EXP] {COMMAND_LIST}`

Executes the command list when `EXP` is true and `VAR` changes.

`when [VAR] at LINE [if EXP] {COMMAND_LIST}`

Executes the command list when `EXP` is true, `VAR` changes, and the debugger encounters `LINE`.

`when in PROCEDURE {COMMAND_LIST}`

Executes the command list upon entering `PROCEDURE`.

`when [VAR] in PROCEDURE [if EXP] {COMMAND_LIST}`

> Executes the specified commands on each line of `PROCEDURE` when `EXP` is true and `VAR` changes. (`EXP` is checked before `VAR`.)

For example:

```
(dbx) when in prnt {print line1.length}
[6] print line1.length in prnt
(dbx) rerun
19                              1
14
19
.
.

17
59
45
12
More (n if no)?
(dbx) delete 6
(dbx) when in prnt {stop}
[7] stop in prnt
(dbx) rerun
[7] stopped at [prnt:52,0x12000fb0] fprintf(stdout,"%3d.(%3d) %s")  2
```

1 Value of `line1.length`.

2 Stops in the procedure `prnt`.

5.8.5 Catching and Ignoring Signals

The `catch` command either lists the signals that `dbx` catches or specifies a signal for `dbx` to catch. If the process encounters a specified signal, `dbx` stops the process.

The `ignore` command either lists the signals that `dbx` does not catch or specifies a signal for `dbx` to add to the ignore list.

The `catch` and `ignore` commands have the following forms:

`catch`	Displays a list of all signals that `dbx` catches.
`catch SIGNAL`	Adds a signal to the catch list.
`ignore`	Displays a list of all signals that `dbx` does not catch.
`ignore SIGNAL`	Removes a signal from the catch list and adds it to the ignore list.

For example:

```
(dbx) catch                                    1
INT QUIT ILL TRAP ABRT EMT FPE BUS SEGV SYS PIPE TERM URG \
STOP TTIN TTOU IO XCPU XFSZ VTALRM PROF WINCH INFO USR1 USR2
(dbx) ignore                                   2
HUP KILL ALRM TSTP CONT CHLD
(dbx) catch kill                               3
(dbx) catch
INT QUIT ILL TRAP ABRT EMT FPE KILL BUS SEGV SYS PIPE TERM URG \
STOP TTIN TTOU IO XCPU XFSZ VTALRM PROF WINCH INFO USR1 USR2
(dbx) ignore
HUP ALRM TSTP CONT CHLD
(dbx)
```

1 Displays the catch list.

2 Displays the ignore list.

3 Adds `KILL` to the catch list and removes `KILL` from the ignore list.

The backslashes in the preceding example represent line continuation. The actual output from `catch` and `ignore` is a single line.

5.9 Examining Program State

When `dbx` is stopped at a breakpoint, the program state can be examined to determine what might have gone wrong. The debugger provides commands for displaying stack traces, variable values, and register values. The debugger also provides commands to display information about the activation levels shown in the stack trace and to move up and down the activation levels (see Section 5.6.2).

5.9.1 Printing the Values of Variables and Expressions

The `print` command displays the values of one or more expressions.

The `printf` command lists information in a specified format and supports all formats of the `printf()` function except strings (`%s`). For a list of formats, see `printf`(3). You can use the `printf` command to see a variable's value in a different number base.

The default command alias list (see Section 5.5.3) provides some useful aliases for displaying the value of variables in different bases — octal (`po`), decimal (`pd`), and hexadecimal (`px`). The default number base is decimal.

You can specify either the real machine register names or the software names from the include file `regdef.h`. A prefix before the register number specifies the type of register; the prefix can be either `$f` or `$r`, as shown in the following list of registers:

Register Name(s)	Register Type
`$f00-$f31`	Floating-point register (1 of 32)
`$r00-$r31`	Machine register (1 of 32)
`$fpcr`	Floating-point control register
`$pc`	Program counter value
`$ps`	Program status register[a]

[a] The program status register is useful only for kernel debugging. For user-level programs, its value is always 8.

You can also specify prefixed registers in the `print` command to display a register value or the program counter. The following commands display the values of machine register 3 and the program counter:

```
(dbx) print $r3
(dbx) print $pc
```

The `print` command has the following forms:

`print EXP1,...,EXPN`

Displays the value of the specified expressions.

`printf "STRING", EXP1,...,EXPN`

Displays the value of the specified expressions in the format specified by the string.

Note

If the expression contains a name that is the same as a `dbx` keyword, you must enclose the name within parentheses. For example, to print `output`, a keyword in the `playback` and `record` commands, specify the name as follows:

```
(dbx) print (output)
```

For example:

```
(dbx) print i
14            1
(dbx) po i
016           2
(dbx) px i
0xe           3
(dbx) pd i
14            4
(dbx)
```

1 Decimal

2 Octal

3 Hexadecimal

4 Decimal

The printregs command displays a complete list of register values; it accepts no arguments. As with the print command, the default base for display by printregs is decimal. To display values in hexadecimal with the printregs command, set the dbx variable $hexints.

For example:

```
(dbx) printregs
$vfp= 4831837712                   $r0_v0=0
$r1_t0=0                           $r2_t1=0
$r3_t2=18446744069416926720        $r4_t3=18446744071613142936
$r5_t4=1                           $r6_t5=0
⋮

$f25= 0.0                          $f26= 0.0
$f27= 2.3873098155006918e-314      $f28= 2.6525639909000367e-314
$f29= 9.8813129168249309e-324      $f30= 2.3872988413145664e-314
$f31= 0.0                          $pc=  4831840840
```

5.9.2 Displaying Activation-Level Information with the dump Command

The dump command displays information about activation levels, including values for all variables that are local to a specified activation level. To see what activation levels are currently active in the program, use the where command to get a stack trace.

The dump command has the following forms:

dump — Displays information about the current activation level.

dump . — Displays information about all activation levels.

dump PROCEDURE — Displays information about the specified procedure (activation level).

For example:

```
(dbx) where
>  0 prnt(pline = 0x11ffffcb8) ["sam.c":52, 0x120000c04]
   1 main(argc = 2, argv = 0x11ffffe08) ["sam.c":45, 0x120000bac]
(dbx) dump
prnt(pline = 0x11ffffcb8) ["sam.c":52, 0x120000c04]
(dbx) dump .
```

```
>  0 prnt(pline = 0x11ffffcb8) ["sam.c":52, 0x120000c04]

   1 main(argc = 2, argv = 0x11ffffe08) ["sam.c":45, 0x120000bac]
line1 = struct {
    string = "#include <stdio.h>"
    length = 19
    linenumber = 0
}
fd = 0x140000158
fname = 0x11ffffe9c = "sam.c"
i = 19
curlinenumber = 1

(dbx) dump main
main(argc = 2, argv = 0x11ffffe08) ["sam.c":45, 0x120000bac]
line1 = struct {
    string = "#include <stdio.h>"
    length = 19
    linenumber = 0
}
fd = 0x140000158
fname = 0x11ffffe9c = "sam.c"
i = 19
curlinenumber = 1
(dbx)
```

5.9.3 Displaying the Contents of Memory

You can display memory contents by specifying the address and the format of the display. Use the following form, with no spaces between the three parts of the command:

address/*count mode*

The *address* portion of the command is the address of the first item to be displayed, *count* is the number of items to be shown, and *mode* indicates the format in which the items are to be displayed. For example:

```
prnt/20i
```

This example displays the contents of 20 machine instructions, beginning at the address of the `prnt` function.

The values for *mode* are shown in Table 5–9.

Table 5–9: Modes for Displaying Memory Addresses

Mode	Display Format
`b`	Displays a byte in octal.
`c`	Displays a byte as a character.
`D`	Displays a long word (64 bits) in decimal.
`d`	Displays a short word (16 bits) in decimal.
`dd`	Displays a word (32 bits) in decimal.
`f`	Displays a single-precision real number.
`g`	Displays a double-precision real number.
`i`	Displays machine instructions.
`O`	Displays a long word in octal.
`o`	Displays a short word in octal.
`oo`	Displays a word (32 bits) in octal.
`s`	Displays a string of characters that ends in a null byte.
`X`	Displays a long word in hexadecimal.
`x`	Displays a short word in hexadecimal.
`xx`	Displays a word (32 bits) in hexadecimal.

The following example shows the output when displaying memory addresses as instructions:

```
(dbx) &prnt/20i
  [prnt:51, 0x120000bf0]  ldah gp, 8193(r27)
  [prnt:51, 0x120000bf4]  lda gp, -25616(gp)
  [prnt:51, 0x120000bf8]  lda sp, -64(sp)
  [prnt:51, 0x120000bfc]  stq r26, 8(sp)
  [prnt:51, 0x120000c00]  stq r16, 16(sp)
  [prnt:52, 0x120000c04]  ldq r16, -32640(gp)
>*[prnt:52, 0x120000c08]  addq r16, 0x38, r16
  [prnt:52, 0x120000c0c]  ldq r17, -32552(gp)
  [prnt:52, 0x120000c10]  ldq r1, 16(sp)
  [prnt:52, 0x120000c14]  ldl r18, 260(r1)
  [prnt:52, 0x120000c18]  ldl r19, 256(r1)
  [prnt:52, 0x120000c1c]  bis r1, r1, r20
  [prnt:52, 0x120000c20]  ldq r27, -32624(gp)
  [prnt:52, 0x120000c24]  jsr r26, (r27), 0x4800030a0
  [prnt:52, 0x120000c28]  ldah gp, 8193(r26)
  [prnt:52, 0x120000c2c]  lda gp, -25672(gp)
  [prnt:54, 0x120000c30]  ldq r16, -32640(gp)
  [prnt:54, 0x120000c34]  addq r16, 0x38, r16
  [prnt:54, 0x120000c38]  ldq r27, -32544(gp)
  [prnt:54, 0x120000c3c]  jsr r26, (r27), 0x480003100
```

5.9.4 Recording and Playing Back Portions of a dbx Session

The `dbx` debugger allows you to capture and replay portions of your input to the program and also portions of its output. Recorded information is written to a file so that you can reuse or re-examine it.

Recording input can be useful for creating command files containing sequences that you want to repeat many times; you can even use recorded input to control `dbx` for purposes such as regression testing. Recording output is useful for capturing large volumes of information that are inconvenient to deal with on the screen, so that you can analyze them later. To look at recorded output later, you can read the saved file directly or you can play it back with `dbx`.

5.9.4.1 Recording and Playing Back Input

The `record input` command records debugger input. The `playback input` command repeats a recorded sequence. The `record input` and `playback input` commands have the following forms:

```
record input [FILE]
```

Begins recording `dbx` commands in the specified file or, if no file is specified, in a file placed in `/tmp` and given a generated name.

```
playback input [FILE]
source [FILE]
```

Executes the commands from the specified file or, if no file is specified, from the temporary file. The two forms are identical in function.

The name given to the temporary file, if used, is contained in the debugger variable `$defaultin`. To display the temporary file name, use the `print` command:

```
(dbx) print $defaultin
```

Use a temporary file when you need to refer to the saved output only during the current debugging session; specify a file name to save information for reuse after you end the current debugging session. Use the `status` command to see whether recording is active. Use the `delete` command to stop recording. Note that these commands will appear in the recording; if you are creating a file for future use, you will probably want to edit the file to remove commands of this type.

Use the `playback input` command to replay the commands recorded with the `record input` command. By default, playback is silent; you do not see the commands as they are played. If the `dbx` variable `$pimode` is set to 1, `dbx` displays commands as they are played back.

The following example records input and displays the resulting file:

```
(dbx) record input                                  1
[2] record input /tmp/dbxtX026963 (0 lines)
(dbx) status
[2] record input /tmp/dbxtX026963 (1 lines)
(dbx) stop in prnt
[3] stop in prnt
(dbx) when i = 19 {stop}
[4] stop ifchanged i = 19
(dbx) delete 2                                      2
(dbx) playback input                                3
[3] stop in prnt
[4] stop ifchanged i = 19
[5] stop in prnt
[6] stop ifchanged i = 19
/tmp/dbxtX026963: 4: unknown event 2                4
(dbx)
```

1 Start recording.

2 Stop recording.

3 Play back the recorded input. As events 3 and 4 are played, they create duplicates of themselves, numbered 5 and 6, respectively.

4 The debugger displays this error message because event 2, the command to begin recording, was deleted when recording was stopped.

The temporary file resulting from the preceding `dbx` commands contains the following text:

```
status
stop in prnt
when i = 19 {stop}
delete 2
```

5.9.4.2 Recording and Playing Back Output

Use the `record output` command to record `dbx` output during a debugging session. To produce a complete record of activity by recording input along with the output, set the `dbx` variable `$rimode`. You can use the debugger's `playback output` command to look at the recorded information, or you can use any text editor.

The `record output` and `playback output` commands have the following forms:

`record output [FILE]`

Begins recording `dbx` output in the specified file or, if no file is specified, in a file placed in `/tmp` and given a generated name.

`playback output [FILE]`

> Displays recorded output from the specified file or, if no file is specified, from the temporary file.

The name given to the temporary file, if used, is contained in the debugger variable `$defaultout`. To display the temporary file name, use the `print` command:

```
(dbx) print $defaultout
```

The `playback output` command works the same as the `cat` command; a display from the `record output` command is identical to the contents of the recording file.

Use a temporary file when you need to refer to the saved output only during the current debugging session; specify a file name to save information for reuse after you end the current debugging session. Use the `status` command to see whether recording is active. Use the `delete` command to stop recording.

The following example shows a sample `dbx` interaction and the output recorded for this interaction in a file named `code`:

```
(dbx) record output code
[3] record output code (0 lines)
(dbx) stop at 25
[4] stop at "sam.c":25
(dbx) run sam.c
[4] stopped at [main:25 ,0x120000a48] if (argc < 2) {
(dbx) delete 3
(dbx) playback output code
[3] record output code (0 lines)
(dbx) [4] stop at "sam.c":25
(dbx) [4] stopped at  [main:25 ,0x120000a48]  if (argc < 2) {
(dbx)
```

5.10 Enabling Core-Dump File Naming

This section explains how to enable the operating system's core-dump file naming feature so that you can preserve multiple core files.

When you enable core-file naming, the system produces core files with names in the following format:

core.*prog-name.host-name.tag*

The name of the core file has four parts separated by periods:

- The literal string `core`.

- Up to 16 characters of the program name, as displayed by the ps command.
- Up to 16 characters of the system's network host name, as derived from the part of the host name that precedes the first dot in the host name format.
- A numeric tag that is assigned to the core file to make it unique among all of the core files generated by a program on a host. The maximum value for this tag, and therefore the maximum number of core files for this program and host, is set by a system configuration parameter (see Section 5.10.1).

  The tag is not a literal version number. The system selects the first available unique tag for the core file. For example, if a program's core files have tags 0, 1, and 3, the system uses tag 2 for the next core file it creates for that program. If the system-configured limit for core-file instances is reached, no further core files are created for that program and host combination. By default, up to 16 versions of a core file can be created.

  To conserve disk space, be sure to remove core files after you have examined them. This is necessary because named core files are not overwritten.

You can enable core-file naming at either the system level (Section 5.10.1) or the individual application level (Section 5.10.2).

5.10.1 Enabling Core-File Naming at the System Level

You can enable core-file naming at the system level by using the dxkerneltuner(8X) (graphical interface) or sysconfig(8) utility. To enable core-file naming, set the enhanced-core-name process subsystem attribute to 1. To limit the number of unique core-file versions that a program can create on a specific host system, set the process subsystem attribute enhanced-core-max-versions to the desired value. For example:

```
proc:

enhanced-core-name = 1
enhanced-core-max-versions = 8
```

The minimum, maximum, and default numbers of versions are 1, 99999, and 16, respectively.

5.10.2 Enabling Core-File Naming at the Application Level

To enable core-file naming at the application level, your program should use the uswitch system call with the UWS_CORE flag set, as in the following example:

```
#include <signal.h>
#include <xsys/uswitch.h>
/*
 * Request enhanced core-file naming for
 * this process, then create a core file.
 */
main()
{
     long uval = uswitch(USC_GET, 0);
     uval = uswitch(USC_SET, uval | USW_CORE);
     if (uval < 0) {
          perror("uswitch");
          exit(1);
     }
     raise(SIGQUIT);
}
```

5.11 Debugging a Running Process

You can use the `dbx` debugger to debug running processes that are started outside the `dbx` environment. It supports the debugging of such processes, both parent and child, by using the `/proc` file system. The debugger can debug running processes only if the `/proc` file system is mounted. If `/proc` is not already mounted, the superuser can mount it with the following command:

```
# mount -t procfs /proc /proc
```

You can add the following entry to the `/etc/fstab` file to mount `/proc` upon booting:

```
/proc           /proc   procfs rw 0 0
```

The `dbx` debugger checks first to see if `/proc` is mounted, but it will still function if this is not the case.

To attach to a running process, use the `dbx` command `attach`, which has the following form:

`attach` *`process-id`*

> The *`process-id`* argument is the process ID of the process you want to attach to.

You can also attach to a process for debugging by using the command-line option –`pid` *`process id`*.

To detach from a running process, use the `dbx` command `detach`, which has the following form:

`detach [`*`process-id`*`]`

> The optional *`process-id`* argument is the process ID of the process you want to detach from. If no argument is given, `dbx` detaches from the current process.

To change from one process to another, use the dbx command switch, which has the following form:

switch *process-id*

> The *process-id* argument is the process ID of the process you want to switch to. You must already have attached to a process before you can switch to it. You can use the alias sw for the switch command.

The attach command first checks to see whether /proc is mounted; dbx gives a warning that tells you what to do if it is not mounted. If /proc is mounted, dbx looks for the process ID in /proc. If the process ID is in /proc, dbx attempts to open the process and issues a stop command. If the process is not there or if the permissions do not allow attaching to it, dbx reports this failure.

When the stop command takes effect, dbx reports the current position, issues a prompt, and waits for user commands. The program probably will not be stopped directly in the user code; it will more likely be stopped in a library or system call that was called by user code.

The detach command deletes all current breakpoints, sets up a “run on last close” flag, and closes (“releases”) the process. The program then continues running if it has not been explicitly terminated inside dbx.

To see a summary of all active processes under the control of dbx, use the plist command, which has the following form:

plist	Displays a list of active processes and their status. Indicates the current process with a marker: -->

5.12 Debugging Multithreaded Applications

The dbx debugger provides three basic commands to assist in the debugging of applications that use threads:

- The tlist command displays a quick list of all threads and where they are currently positioned in the program. This command accepts no arguments.

  Using the tlist command, you can see all of the threads, with their IDs, that are currently in your program.

- The tset command sets the current thread. The debugger maintains one thread as the current thread; this thread is the one that hits a breakpoint or receives a signal that causes it to stop and relinquish control to dbx.

  Use tset to choose a different thread as the current thread so that you can examine its state with the usual dbx commands. Note that the selected thread remains the current thread until you enter another tset command. Note also that the continue, step, or next commands might be inappropriate for a given thread if it is blocked or waiting to join with another thread.

- The tstack command lists the stacks of all threads in your application. It is similar to the where command and, like where, takes an optional numeric argument to limit the number of stack levels displayed.

The tset and tstack commands have the following forms:

tset [EXP]

Choose a thread to be the current thread. The EXP argument is the hexadecimal ID of the desired thread.

tstack [EXP]

Display stack traces for all threads. If EXP is specified, dbx displays only the top EXP levels of the stacks; otherwise, the entire stacks are displayed.

If the POSIX Threads Library product is installed on your system, you can gain access to the POSIX Threads Library pthread debugger by issuing a call cma_debug() command within your dbx session. The pthread debugger can provide a great deal of useful information about the threads in your program. For information on using the pthread debugger, enter a help command at its debug> prompt.

A sample threaded program, twait.c, is shown in Example 12–1. The following example shows a dbx session using that program. Long lines in this example have all been folded at 72 characters to represent display on a narrow terminal.

```
% dbx twait
dbx version 3.11.6
Type ’help’ for help.

main:  50  pthread_t       me = pthread_self(), timer_thread;
(dbx) stop in do_tick
[2] stop in do_tick
(dbx) stop at 85
[3] stop at "twait.c":85
(dbx) stop at 35
[4] stop at "twait.c":35
(dbx) run
1: main thread starting up
1: exit lock initialized
1: exit lock obtained
1: exit cv initialized
1: timer_thread 2 created
1: exit lock released
[2] thread 0x81c62e80 stopped at   [do_tick:21 ,0x12000730c]    pthread_
t        me = pthread_self();
(dbx) tlist
thread 0x81c623a0 stopped at   [msg_receive_trap:74 +0x8,0x3ff808edf04]
 Source not available
thread 0x81c62e80 stopped at   [do_tick:21 ,0x12000730c]        pthread_
t        me = pthread_self();
```

```
(dbx) where
>  0 do_tick(argP = (nil)) ["twait.c":21, 0x12000730c]
   1 cma__thread_base(0x0, 0x0, 0x0, 0x0, 0x0) ["../../../../../src/usr/
ccs/lib/DECthreads/COMMON/cma_thread.c":1441, 0x3ff80931410]
(dbx) tset 0x81c623a0
thread 0x81c623a0 stopped at   [msg_receive_trap:74 +0x8,0x3ff808edf04]
 Source not available
(dbx) where
>  0 msg_receive_trap(0x3ff8087b8dc, 0x3ffc00a2480, 0x3ff8087b928, 0x181
57f0d0d, 0x3ff8087b68c) ["/usr/build/osf1/goldos.bld/export/alpha/usr/in
clude/mach/syscall_sw.h":74, 0x3ff808edf00]
   1 msg_receive(0x61746164782e, 0x3ffc009a420, 0x3ffc009a420, 0x3c20, 0
xe0420) ["../../../../../src/usr/ccs/lib/libmach/msg.c":95, 0x3ff808e474
4]
   2 cma__vp_sleep(0x280187f578, 0x3990, 0x7, 0x3ffc1032848, 0x0) ["../.
./../../../src/usr/ccs/lib/DECthreads/COMMON/cma_vp.c":1471, 0x3ff809375
cc]
   3 cma__dispatch(0x7, 0x3ffc1032848, 0x0, 0x3ffc100ee08, 0x3ff80917e3c
) ["../../../../../src/usr/ccs/lib/DECthreads/COMMON/cma_dispatch.c":967
, 0x3ff80920e48]
   4 cma__int_wait(0x11ffff228, 0x140009850, 0x3ffc040cdb0, 0x5, 0x3ffc0
014c00) ["../../../../../src/usr/ccs/lib/DECthreads/COMMON/cma_condition
.c":2202, 0x3ff80917e38]
   5 cma_thread_join(0x11ffff648, 0x11ffff9f0, 0x11ffff9e8, 0x60aaec4, 0
x3ff8000cf38) ["../../../../../src/usr/ccs/lib/DECthreads/COMMON/cma_thr
ead.c":825, 0x3ff80930a58]
   6 pthread_join(0x140003110, 0x40002, 0x11ffffa68, 0x3ffc040cdb0, 0x0)
 ["../../../../../src/usr/ccs/lib/DECthreads/COMMON/cma_pthread.c":2193,
 0x3ff809286c8]
   7 main() ["twait.c":81, 0x12000788c]
(dbx) tlist
thread 0x81c623a0 stopped at   [msg_receive_trap:74 +0x8,0x3ff808edf04]
 Source not available
thread 0x81c62e80 stopped at   [do_tick:21 ,0x12000730c]        pthread_
t        me = pthread_self();
(dbx) tset 0x81c62e80
thread 0x81c62e80 stopped at   [do_tick:21 ,0x12000730c]        pthread_
t        me = pthread_self();
(dbx) cont
2: timer thread starting up, argP=0x0
[4] thread 0x81c62e80 stopped at   [do_tick:35 ,0x120007430]    printf("
%d: wait for next tick\n", THRID(&me));
(dbx) cont
2: wait for next tick
2: TICK #1
[4] thread 0x81c62e80 stopped at   [do_tick:35 ,0x120007430]    printf("
%d: wait for next tick\n", THRID(&me));
(dbx) tstack
Thread 0x81c623a0:
>  0 msg_receive_trap(0x3ff8087b8dc, 0x3ffc00a2480, 0x3ff8087b928, 0x181
57f0d0d, 0x3ff8087b68c) ["/usr/build/osf1/goldos.bld/export/alpha/usr/in
clude/mach/syscall_sw.h":74, 0x3ff808edf00]
   1 msg_receive(0x61746164782e, 0x3ffc009a420, 0x3ffc009a420, 0x3c20, 0
xe0420) ["../../../../../src/usr/ccs/lib/libmach/msg.c":95, 0x3ff808e474
4]
   2 cma__vp_sleep(0x280187f578, 0x3990, 0x7, 0x3ffc1032848, 0x0) ["../.
./../../../src/usr/ccs/lib/DECthreads/COMMON/cma_vp.c":1471, 0x3ff809375
cc]
   3 cma__dispatch(0x7, 0x3ffc1032848, 0x0, 0x3ffc100ee08, 0x3ff80917e3c
) ["../../../../../src/usr/ccs/lib/DECthreads/COMMON/cma_dispatch.c":967
, 0x3ff80920e48]
   4 cma__int_wait(0x11ffff228, 0x140009850, 0x3ffc040cdb0, 0x5, 0x3ffc0
014c00) ["../../../../../src/usr/ccs/lib/DECthreads/COMMON/cma_condition
.c":2202, 0x3ff80917e38]
```

```
   5 cma_thread_join(0x11ffff648, 0x11ffff9f0, 0x11ffff9e8, 0x60aaec4, 0
x3ff8000cf38) ["../../../../../src/usr/ccs/lib/DECthreads/COMMON/cma_thr
ead.c":825, 0x3ff80930a58]
   6 pthread_join(0x140003110, 0x40002, 0x11ffffa68, 0x3ffc040cdb0, 0x0)
 ["../../../../../src/usr/ccs/lib/DECthreads/COMMON/cma_pthread.c":2193,
 0x3ff809286c8]
   7 main() ["twait.c":81, 0x12000788c]
Thread 0x81c62e80:
>  0 do_tick(argP = (nil)) ["twait.c":35, 0x120007430]
   1 cma__thread_base(0x0, 0x0, 0x0, 0x0, 0x0) ["../../../../../src/usr/
ccs/lib/DECthreads/COMMON/cma_thread.c":1441, 0x3ff80931410]
More (n if no)?
(dbx) tstack 3
Thread 0x81c623a0:
>  0 msg_receive_trap(0x3ff8087b8dc, 0x3ffc00a2480, 0x3ff8087b928, 0x181
57f0d0d, 0x3ff8087b68c) ["/usr/build/osf1/goldos.bld/export/alpha/usr/in
clude/mach/syscall_sw.h":74, 0x3ff808edf00]
   1 msg_receive(0x61746164782e, 0x3ffc009a420, 0x3ffc009a420, 0x3c20, 0
xe0420) ["../../../../../src/usr/ccs/lib/libmach/msg.c":95, 0x3ff808e474
4]
   2 cma__vp_sleep(0x280187f578, 0x3990, 0x7, 0x3ffc1032848, 0x0) ["../.
./../../../src/usr/ccs/lib/DECthreads/COMMON/cma_vp.c":1471, 0x3ff809375
cc]
Thread 0x81c62e80:
>  0 do_tick(argP = (nil)) ["twait.c":35, 0x120007430]
   1 cma__thread_base(0x0, 0x0, 0x0, 0x0, 0x0) ["../../../../../src/usr/
ccs/lib/DECthreads/COMMON/cma_thread.c":1441, 0x3ff80931410]
(dbx) cont
2: wait for next tick
2: TICK #2
[4] thread 0x81c62e80 stopped at   [do_tick:35 ,0x120007430]    printf("
%d: wait for next tick\n", THRID(&me));
(dbx) assign ticks = 29
29
(dbx) cont
2: wait for next tick
2: TICK #29
[4] thread 0x81c62e80 stopped at   [do_tick:35 ,0x120007430]    printf("
%d: wait for next tick\n", THRID(&me));
(dbx) cont
2: wait for next tick
2: TICK #30
2: exiting after #31 ticks
1: joined with timer_thread 2
[3] thread 0x81c623a0 stopped at   [main:85 ,0x1200078ec]       if (errn
o != 0) printf("errno 7 = %d\n",errno);
(dbx) tlist
thread 0x81c623a0 stopped at   [main:85 ,0x1200078ec]   if (errno != 0)
printf("errno 7 = %d\n",errno);
thread 0x81c62e80 stopped at   [msg_rpc_trap:75 +0x8,0x3ff808edf10]
 Source not available
(dbx) cont

Program terminated normally

(dbx) tlist
(dbx) quit
```

5.13 Debugging Multiple Asynchronous Processes

The dbx debugger can debug multiple simultaneous asynchronous processes. While debugging asynchronous processes, dbx can display status and accept commands asynchronously. When running asynchronously, the debugger might exhibit confusing behavior because a running process can display output on the screen while you are entering commands to examine a different process that is stopped.

The debugger automatically enters asynchronous mode in either of the following circumstances:

- You command it to attach to a new process while a previous process is still attached.
- The process to which dbx is attached forks off a child process, and the debugger automatically attaches to the child process without detaching from the parent.

The debugger uses several predefined variables to define the behavior of asynchronous debugging. (See Table 5–8.) The variable $asynch_interface can be viewed as a counter that is incremented by 1 when a new process is attached and decremented by 1 when a process terminates or is detached. The default value is 0.

When $asynch_interface has a positive nonzero value, asynchronous debugging is enabled; when the variable is 0 (zero) or negative, asynchronous debugging is disabled. To prevent dbx from entering asynchronous mode, set the $asynch_interface variable to a negative value. (Note that disabling asynchronous mode might make debugging more difficult if a parent is waiting on a child that is stopped.)

When a process executes a fork( ) or vfork( ) call to spawn a child process, dbx attaches to the child process and automatically enters asynchronous mode (if permitted by $asynch_interface). The default behavior is to stop the child process right after the fork. You can change this default by setting the variable $stop_on_fork to 0; in this case, dbx will attach to the child process but not stop it.

The dbx debugger attempts to apply a degree of intelligence to the handling of forks by filtering out many of the fork calls made by various system and library calls. If you want to stop the process on these forks also, you can set the predefined variable $stop_all_forks to 1. This variable's default value is 0. Stopping on all forks can be particularly useful when you are debugging a library routine.

You can use the debugger's plist and switch commands to monitor and switch between processes.

5.14 Sample Program

Example 5–1 is the sample C program (sam.c) that is referred to in examples throughout this chapter.

Example 5–1: Sample Program Used in dbx Examples

```
#include <stdio.h>
struct line {
    char string[256];
    int length;
    int linenumber;
};

typedef struct line LINETYPE;

void prnt();

main(argc,argv)
    int argc;
    char **argv;
{
    LINETYPE line1;
    FILE *fd;
    extern FILE *fopen();
    extern char *fgets();
    extern char *getenv();
    char *fname;
    int i;
    static curlinenumber=0;

    if (argc < 2) {
        if((fname = getenv("TEXT")) == NULL || *fname == ’ ’) {
            fprintf(stderr, "Usage: sam filename\n");
            exit(1);
        }
    } else
        fname = argv[1];

    fd = fopen(fname,"r");
    if (fd == NULL) {
        fprintf(stderr, "cannot open %s\n",fname);
        exit(1);
    }

    while(fgets(line1.string, sizeof(line1.string), fd) != NULL){
        i=strlen(line1.string);
        if (i==1 && line1.string[0] == ’\n’)
            continue;
        line1.length = i;
        line1.linenumber = curlinenumber++;
        prnt(&line1);
    }
}
```

Example 5–1: Sample Program Used in dbx Examples (cont.)

```
void prnt(pline)
LINETYPE *pline;
{
    fprintf(stdout,"%3d. (%3d) %s",
        pline->linenumber, pline->length, pline->string);
    fflush(stdout);
}
```

6

Checking C Programs with lint

You can use the `lint` program to check your C programs for potential coding problems. The `lint` program checks a program more carefully than some C compilers, and displays messages that point out possible problems. Some of the messages require corrections to the source code; others are only informational messages and do not require corrections.

This chapter addresses the following topics:

- Syntax of the `lint` command (Section 6.1)
- Program flow checking (Section 6.2)
- Data type checking (Section 6.3)
- Variable and function checking (Section 6.4)
- Checking the use of variables before they are initialized (Section 6.5)
- Migration checking (Section 6.6)
- Portability checking (Section 6.7)
- Checking for coding errors and coding style differences (Section 6.8)
- Increasing table sizes for large programs (Section 6.9)
- Creating a `lint` library (Section 6.10)
- Understanding `lint` error messages (Section 6.11)
- Using warning class options to suppress `lint` messages (Section 6.12)
- Generating function prototypes for compile-time detection of syntax errors (Section 6.13)

See `lint`(1) for a complete list of `lint` options.

6.1 Syntax of the lint Command

The `lint` command has the following syntax:

lint [*options*] [*file* ...]

`options`

Options to control `lint` checking operations.

The cc driver options, –std, –std0, and –std1, are available as options to lint. These options affect the parsing of the source as well as the selection of the lint library to use. Selecting either the –std or –std1 options turns on ANSI parsing rules in lint.

When you use the –MA lint option, –std1 is used for the C preprocessing phase and _ANSI_C_SOURCES is defined using the –D preprocessor option. The following table describes the action lint takes for each option:

lint Option	Preprocessor Switch	lint Parsing	lint Library
–MA	–std1 and –D_ANSI_C_SOURCE	ANSI	llib-lansi.ln
–std	–std	ANSI	llib-lcstd.ln
–std1	–std1	ANSI	llib-lcstd.ln
–std0	–std0	EXTD[a]	llib-lc.ln

[a] EXTD is Extended C language, also know as K&R C.

file

The name of the C language source file for lint to check. The name must have one of the following suffixes:

Suffix	Description
.c	C source file
.i	File produced by the C preprocessor (cpp)
.ln	lint library file

Note that lint library files are the result of a previous invocation of the lint program with either the -c or -o option. They are analogous to the .o files produced by the cc command when it is given a .c file as input. The ability to specify lint libraries as input to the lint program facilitates intermodule interface checking in large applications. Adding rules that specify the construction of lint libraries to their makefiles can make building such applications more efficient. See Section 6.10 for a discussion on how to create a lint library.

You can also specify as input a lint library that resides in one of the system's default library search directories by using the -l*x* option. The library name must have the following form:

llib-l*libname*.ln

By default, the lint program appends the extended C (K&R C) lint library (llib-lc.ln) to the list of files specified on the command line. If the −std or −std1 option is used, it appends the standard C lint library (llib-lcstd.ln) instead.

The following additional libraries are included with the system:

Library	Description	Specify As
crses	Checks curses library call syntax	−lcrses
m	Checks math library call syntax	−lm
port	Checks for portability with other systems	−p (not −lport)
ansi	Enforces ANSI C standard rules	−MA (not −lansi)

If you specify no options on the command line, the lint program checks the specified C source files and writes messages about any of the following coding problems that it finds:

- Loops that are not entered and exited normally
- Data types that are not used correctly
- Functions that are not used correctly
- Variables that are not used correctly
- Coding techniques that could cause problems if a program is moved to another system
- Nonstandard coding practices and style differences that could cause problems

The lint program also checks for syntax errors in statements in the source programs. Syntax checking is always done and is not influenced by any options that you specify on the lint command.

If lint does not report any errors, the program has correct syntax and will compile without errors. Passing that test, however, does not mean that the program will operate correctly or that the logic design of the program is accurate.

See Section 6.10 for information on how to create your own lint library.

6.2 Program Flow Checking

The lint program checks for dead code, that is, parts of a program that are never executed because they cannot be reached. It writes messages about statements that do not have a label but immediately follow statements that change the program flow, such as goto, break, continue, and return.

The lint program also detects and writes messages for loops that cannot be entered at the top. Some programs that include this type of loop may produce correct results; however, this type of loop can cause problems.

The lint program does not recognize functions that are called but can never return to the calling program. For example, a call to exit may result in code that cannot be reached, but lint does not detect it.

Programs generated by yacc and lex may have hundreds of break statements that cannot be reached. The lint program normally writes an error message for each of these break statements. To eliminate the extraneous code associated with these break statements, use the -O option to the cc command when compiling the program. Use the –b option with the lint program to prevent it from writing these messages when checking yacc and lex output code. (For information on yacc and lex, see the *Programming Support Tools* manual.)

6.3 Data Type Checking

The lint program enforces the type-checking rules of the C language more strictly than the compiler does. In addition to the checks that the compiler makes, lint checks for potential data type errors in the following areas:

- Binary operators and implied assignments
- Structures and unions
- Function definition and uses
- Enumerators
- Type-checking control
- Type casts

Details on each of these potential problem areas are provided in the sections that follow.

6.3.1 Binary Operators and Implied Assignments

The C language allows the following data types to be mixed in statements, and the compiler does not indicate an error when they are mixed:

```
char
short
int
long
unsigned
float
double
```

The C language automatically converts data types within this group to provide the programmer with more flexibility in programming. This flexibility, however, means that the programmer, not the language, must ensure that the data type mixing produces the desired result.

You can mix these data types when using them in the following ways (in the examples, alpha is type char and num is type int):

- Operands on both sides of an assignment operator, for example:

  ```
  alpha = num;   /* alpha converts to int */
  ```

- Operands in a conditional expression, for example:

  ```
  value=(alpha < num) ? alpha : num;
  /* alpha converts to int */
  ```

- Operands on both sides of a relational operator, for example:

  ```
  if( alpha != num )  /* alpha converts to int */
  ```

- The type of an argument in a return statement is converted to the type of the value that the function returns, for example:

  ```
  funct(x)        /* returns an integer */
  {
    return( alpha );
  }
  ```

The data types of pointers must agree exactly, except that you can mix arrays of any type with pointers to the same type.

6.3.2 Structures and Unions

The lint program checks structure operations for the following requirements:

- The left operand of the structure pointer operator (->) must be a pointer to a structure.
- The left operand of the structure member operator (.) must be a structure.
- The right operand of these operators must be a member of the same structure.

The lint program makes similar checks for references to unions.

6.3.3 Function Definition and Uses

The lint program applies strict rules to function argument and return value matching. Arguments and return values must agree in type, with the following exceptions:

- You can match arguments of type `float` with arguments of type `double`.
- You can match arguments within the following types:

```
char
short
int
unsigned
```

- You can match pointers with the associated arrays.

6.3.4 Enumerators

The `lint` program checks enumerated data type variables to ensure that they meet the following requirements:

- Enumerator variables or members of an enumerated type are not mixed with other types or other enumerator variables.
- The enumerated data type variables are only used in the following areas:

  Assignment (=)
  Initialization
  Equivalence (==)
  Not equivalence (!=)
  Function arguments
  Return values

6.3.5 Type Casts

Type casts in the C language allow the program to treat data of one type as if it were data of another type. The `lint` program can check for type casts and write a message if it finds one.

The `-wp` and `-h` options for the `lint` command line control the writing of warning messages about casts. If neither of these options are used, `lint` produces warning messages about casts that may cause portability problems.

In migration checking mode, `-Qc` suppresses cast warning messages (see Section 6.6).

6.4 Variable and Function Checking

The `lint` program checks for variables and functions that are declared in a program but not used. The `lint` program checks for the following errors in the use of variables and functions:

- Functions that return values inconsistently
- Functions that are defined but not used

- Arguments to a function call that are not used
- Functions that can return either with or without values
- Functions that return values that are never used
- Programs that use the value of a function when the function does not return a value

Details on each of these potential problem areas are provided in the sections that follow.

6.4.1 Inconsistent Function Return

If a function returns a value under one set of conditions but not under another, you cannot predict the results of the program. The `lint` program checks functions for this type of behavior. For example, if both of the following statements are in a function definition, a program calling the function may or may not receive a return value:

```
return(expr);
⋮

return;
```

These statements cause the `lint` program to write the following message to point out the potential problem:

```
function name has return(e); and return
```

The `lint` program also checks functions for returns that are caused by reaching the end of the function code (an implied return). For example, in the following part of a function, if `a` tests false, `checkout` calls `fix_it` and then returns with no defined return value:

```
checkout (a)
{
        if (a) return (3);
        fix_it ();
}
```

These statements cause the `lint` program to write the following message:

```
function checkout has return(e); and return
```

If `fix_it`, like `exit`, never returns, `lint` still writes the message even though nothing is wrong.

6.4.2 Function Values That Are Not Used

The `lint` program checks for cases in which a function returns a value and the calling program may not use the value. If the value is never used, the function definition may be inefficient and should be examined to determine whether it should be modified or eliminated. If the value is sometimes used, the function may be returning an error code that the calling program does not check.

6.4.3 Disabling Function-Related Checking

To prevent `lint` from checking for problems with functions, specify one or more of the following options to the `lint` command:

`-x` Do not check for variables that are declared in an `extern` statement but never used.

`-v` Do not check for arguments to functions that are not used, except for those that are also declared as register arguments.

`-u` Do not check for functions and external variables that are either used and not defined or defined and not used. Use this option to eliminate useless messages when you are running `lint` on a subset of files of a larger program. (When using `lint` with some, but not all, files that operate together, many of the functions and variables defined in those files may not be used. Also, many functions and variables defined elsewhere may be used.)

You can also place directives in the program to control checking:

- To prevent `lint` from warning about unused function arguments, add the following directive to the program before the function definition:

  ```
  /*ARGSUSED*/
  ```

- To prevent `lint` from writing messages about variable numbers of arguments in calls to a function, add the following directive before the function definition:

  ```
  /*VARARGSn*/
  ```

  To check the first several arguments and leave the later arguments unchecked, add a digit (*n*) to the end of the VARARGS directive to give the number of arguments that should be checked, such as:

  ```
  /*VARARGS2*/
  ```

  When `lint` reads this directive, it checks only the first two arguments.

- To suppress complaints about unused functions and function arguments in an entire file, place the following directive at the beginning of the file:

  ```
  /*LINTLIBRARY*/
  ```

  This is equivalent to using the `–v` and `–x` options.

- To permit a standard prototype checking library to be formed from header files by making function prototype declarations appear as function definitions, use the following directive:

  ```
  /*LINTSTDLIB[_filename]*/
  ```

The /*LINTSTDLIB*/ directive implicitly activates the functions of the /*NOTUSED*/ and /*LINTLIBRARY*/ directives to reduce warning noise levels. When a file is referenced (*filename*), only prototypes in that file are expanded. Multiple /*LINTSTDLIB_*filename* */ statements are allowed. (See Section 6.10.1 for more details on the use of /*LINTSTDLIB*/ directives.)

- To suppress warnings about all used but undefined external symbols and functions that are subsequently encountered in the file, use the following directive:

  ```
  /*NOTDEFINED*/
  ```

- To suppress comments about unreachable code, use the following directive:

  ```
  /*NOTREACHED*/
  ```

  When placed at appropriate points in a program (typically immediately following a return, break, or continue statement), the /*NOTREACHED*/ directive stops comments about unreachable code. Note that lint does not recognize the exit function and other functions that may not return.

- To suppress warnings about all unused external symbols, functions, and function parameters that are subsequently encountered in the file, use the following directive:

  ```
  /*NOTUSED*/
  ```

  The /*NOTUSED*/ directive is similar to the /*LINTLIBRARY*/ directive, although /*NOTUSED*/ also applies to external symbols.

6.5 Checking on the Use of Variables Before They Are Initialized

The lint program checks for the use of a local variable (auto and register storage classes) before a value has been assigned to it. Using a variable with an auto (automatic) or register storage class also includes taking the address of the variable. This is necessary because the program can use the variable (through its address) any time after it knows the address of the variable. Therefore, if the program does not assign a value to the variable before it finds the address of the variable, lint reports an error.

Because lint only checks the physical order of the variables and their usage in the file, it may write messages about variables that are initialized properly (in execution sequence).

The lint program recognizes and writes messages about:

- Initialized automatic variables

- Variables that are used in the expression that first sets them
- Variables that are set and never used

Note

> The Tru64 UNIX operating system initializes `static` and `extern` variables to zero. Therefore, `lint` assumes that these variables are set to zero at the start of the program and does not check to see if they have been assigned a value when they are used. When developing a program for a system that does not do this initialization, ensure that the program sets `static` and `extern` variables to an initial value.

6.6 Migration Checking

Use `lint` to check for all common programming techniques that might cause problems when migrating programs from 32-bit operating systems to the Tru64 UNIX operating system. The `–Q` option provides support for checking ULTRIX and DEC OSF/1 Version 1.0 programs that you are migrating to 64-bit systems.

Because the `–Q` option disables checking for most other programming problems, use this option only for migration checking. Suboptions are available to suppress specific categories of checking. For example, entering `–Qa` suppresses the checking of pointer alignment problems. You can enter more than one suboption with the `–Q` option, for example, `–QacP` to suppress checking for pointer alignment problems, problematic type casts, and function prototype checks, respectively. For more information about migration checking, see `lint`(1).

6.7 Portability Checking

Use `lint` to help ensure that you can compile and run C programs using different C language compilers and other systems.

The following sections indicate areas to check before compiling the program on another system. Checking only these areas, however, does not guarantee that the program will run on any system.

Note

> The `llib-port.ln` library is brought in by using the `–p` option, not by using the `–lport` option.

6.7.1 Character Uses

Some systems define characters in a C language program as signed quantities with a range from –128 to 127; other systems define characters as positive values. The `lint` program checks for character comparisons or assignments that may not be portable to other systems. For example, the following fragment may work on one system but fail on systems where characters always take on positive values:

```
char c;
 .
 .
 .
if( ( c = getchar() ) <0 )...
```

This statement causes the `lint` program to write the following message:

```
nonportable character comparison
```

To make the program work on systems that use positive values for characters, declare `c` as an integer because `getchar` returns integer values.

6.7.2 Bit Field Uses

Bit fields may produce problems when a program is transferred to another system. Bit fields may be signed quantities on the new system. Therefore, when constant values are assigned to a bit field, the field may be too small to hold the value. To make this assignment work on all systems, declare the bit field to be of type `unsigned` before assigning values to it.

6.7.3 External Name Size

When changing from one type of system to another, be aware of differences in the information retained about external names during the loading process:

- The number of characters allowed for external names can vary.
- Some programs that the compiler command calls and some of the functions that your programs call can further limit the number of significant characters in identifiers. (In addition, the compiler adds a leading underscore to all names and keeps uppercase and lowercase characters separate.)
- On some systems, uppercase or lowercase may not be important or may not be allowed.

When transferring from one system to another, you should always take the following steps to avoid problems with loading a program:

1. Review the requirements of each system.
2. Run `lint` with the `-p` option.

The -p option tells lint to change all external symbols to lowercase and limit them to six characters while checking the input files. The messages produced identify the terms that may need to be changed.

6.7.4 Multiple Uses and Side Effects

Be careful when using complicated expressions because of the following considerations:

- The order in which complex expressions are evaluated differs in many C compilers.
- Function calls that are arguments of other functions may not be treated the same as ordinary arguments.
- Operators such as assignment, increment, and decrement may cause problems when used on different systems.

The following situations demonstrate the types of problems that can result from these differences:

- If any variable is changed by a side effect of one of the operators and is also used elsewhere in the same expression, the result is undefined.
- The evaluation of the variable years in the following printf statement is confusing because on some machines years is incremented before the function call and on other machines it is incremented after the function call:

  ```
  printf( "%d %d\n", ++years, amort( interest, years ) );
  ```

- The lint program checks for simple scalar variables that may be affected by evaluation order problems, such as in the following statement:

  ```
  a[i]=b[i++];
  ```

  This statement causes the lint program to write the following message:

  ```
  warning: i evaluation order undefined
  ```

6.8 Checking for Coding Errors and Coding Style Differences

Use lint to detect possible coding errors, and to detect differences from the coding style that lint expects. Although coding style is mainly a matter of individual taste, examine each difference to ensure that the difference is both needed and accurate. The following sections indicate the types of coding and style problems that lint can find.

6.8.1 Assignments of Long Variables to Integer Variables

If you assign variables of type `long` to variables of type `int`, the program may not work properly. The `long` variable is truncated to fit in the integer space and data may be lost.

An error of this type frequently occurs when a program that uses more than one `typedef` is converted to run on a different system.

To prevent `lint` from writing messages when it detects assignments of `long` variables to `int` variables, use the `-a` option.

6.8.2 Operator Precedence

The `lint` program detects possible or potential errors in operator precedence. Without parentheses to show order in complex sequences, these errors can be hard to find. For example, the following statements are not clear:

```
if(x&077==0). . .   /* evaluated as: if(x & (077 == 0) ) */
                    /* should be: if( (x & 077) == 0) */

x<<2+40             /* evaluated as: x <<(2+40) */
                    /* should be: (x<<2) + 40 */
                    /* shift x left 42 positions */
```

Use parentheses to make the operation more clearly understood. If you do not, `lint` issues a message.

6.8.3 Conflicting Declarations

The `lint` program writes messages about variables that are declared in inner blocks in ways that conflict with their use in outer blocks. This practice is allowed, but may cause problems in the program.

Use the `-h` option with the `lint` program to prevent `lint` from checking for conflicting declarations.

6.9 Increasing Table Size

The `lint` command provides the `–N` option and related suboptions to allow you to increase the size of various internal tables at run time if the default values are not enough for your program. These tables include:

- Symbol table
- Dimension table
- Local type table
- Parse tree

These tables are dynamically allocated by the lint program. Using the –N option on large source files can improve performance.

6.10 Creating a lint Library

For programming projects that define additional library routines, you can create an additional lint library to check the syntax of the programs. Using this library, the lint program can check the new functions in addition to the standard C language functions. To create a new lint library, follow these steps:

1. Create an input file that defines the new functions.
2. Process the input file to create the lint library file.
3. Run lint using the new library.

The following sections describe these steps.

6.10.1 Creating the Input File

The following example shows an input file that defines three additional functions for lint to check:

```
/*LINTLIBRARY*/

#include <dms.h>

int  dmsadd( rmsdes, recbuf, reclen )
               int rmsdes;
               char *recbuf;
               unsigned reclen;
             { return 0; }
int  dmsclos( rmsdes)
               int rmsdes;
             { return 0; }
int  dmscrea( path, mode, recfm, reclen )
               char *path;
               int mode;
               int recfm;
               unsigned reclen;
             { return 0; }
```

The input file is a text file that you create with an editor. It consists of:

- A directive to tell the cpp program that the following information is to be made into a library of lint definitions:

  ```
  /*LINTLIBRARY*/
  ```

- A series of function definitions that define:

- The type of the function (int in the example)
- The name of the function
- The parameters that the function expects
- The types of the parameters
- The value that the function returns

Alternatively, you can create a lint library file from function prototypes. For example, assume that the dms.h file includes the following prototypes:

```
int dmsadd(int,
           char*,
           unsigned);
int dmsclose(int);
int dmscrea(char*,
            int,
            int,
            unsigned);
```

In this case, the input file contains the following:

```
/*LINTSTDLIB*/
#include <dms.h>
```

In the case where a header file may include other headers, the LINTSTDLIB command can be restricted to specific files:

```
/*LINTSTDLIB_dms.h*/
```

In this case, only prototypes declared in dms.h will be expanded. Multiple LINTSTDLIB commands can be included.

In all cases, the name of the input file must have the prefix llib-l. For example, the name of the sample input file created in this section could be llib-ldms. When choosing the name of the file, ensure that it is not the same as any of the existing files in the /usr/ccs/lib directory.

6.10.2 Creating the lint Library File

The following command creates a lint library file from the input file described in the previous section:

```
% lint [options] -c llib_ldms.c
```

This command tells lint to create a lint library file, llib-ldms.ln, using the file llib-ldms.c as input. To use llib-ldms.ln as a system lint library (that is, a library specified in the -l*x* option of the lint command), move it to /usr/ccs/lib. Use the −std or −std1 option to use ANSI preprocessing rules to build the library.

6.10.3 Checking a Program with a New Library

To check a program using a new library, use the `lint` command with the following format:

`lint -l`*`pgm filename.c`*

The variable *`pgm`* represents the identifier for the library, and the variable *`filename.c`* represents the name of the file containing the C language source code that is to be checked. If no other options are specified, the `lint` program checks the C language source code against the standard `lint` library in addition to checking it against the indicated special `lint` library.

6.11 Understanding lint Error Messages

Although most error messages produced by `lint` are self-explanatory, certain messages may be misleading without additional explanation. Usually, once you understand what a message means, correcting the error is straightforward. The following is a list of the more ambiguous `lint` messages:

`constant argument to NOT`

A constant is used with the NOT operator (!). This is a common coding practice and the message does not usually indicate a problem. The following program demonstrates the type of code that can generate this message:

```
% cat x.c
#include <stdio.h>
#define SUCCESS    0

main()
{
   int value = !SUCCESS;

   printf("value = %d\n", value);
   return 0;
}
% lint -u x.c
"x.c", line 7: warning: constant argument to NOT
% ./x
value = 1
%
```

The program runs as expected, even though `lint` complains.

Recommended Action: Suppress these `lint` warning messages by using the `-wC` option.

```
constant in conditional context
```

A constant is used where a conditional is expected. This problem occurs often in source code due to the way in which macros are encoded. For example:

```
typedef struct _dummy_q {
  int lock;
  struct _dummy_q *head, *tail;
} DUMMY_Q;

#define QWAIT   1
#define QNOWAIT 0
#define DEQUEUE(q, elt, wait)    1           \
        for (;;) {
            simple_lock(&(q)->lock);
        if (queue_empty(&(q)->head))
            if (wait) {          1           \
                assert(q);
                simple_unlock(&(q)->lock);
                continue;
            } else
                *(elt) = 0;
        else
                  dequeue_head(&(q)->head);
                  simple_unlock(&(q)->lock);
        break;
    }

int doit(DUMMY_Q *q, int *elt)
{
  DEQUEUE(q, elt, QNOWAIT);
}
```

1 The `QWAIT` or `QNOWAIT` option is passed as the third argument (`wait`) and is later used in the `if` statement. The code is correct, but `lint` issues the warning because constants used in this way are normally unnecessary and often generate wasteful or unnecessary instructions.

Recommended Action: Suppress these `lint` warning messages by using the `-wC` option.

```
conversion from long may lose accuracy
```

A signed `long` is copied to a smaller entity (for example, an `int`). This message is not necessarily misleading; however, if it occurs frequently, it may or may not indicate a coding problem, as shown in the following example:

```
long BuffLim = 512;        1

void foo (buffer, size)
char *buffer;
int size;
{
register int count;
register int limit = size < (int)BufLimit ? size : (int)BufLim;  1
```

1 The `lint` program reports the conversion error, even though the appropriate `(int)` cast exists.

Recommended Action: Review code sections for which `lint` reports this message, or suppress the message by using the `-wl` option.

`declaration is missing declarator`

A line in the declaration section of the program contains just a semicolon (;). Although you would not deliberately write code like this, it is easy to inadvertently generate such code by using a macro followed by a semicolon. If, due to conditionalization, the macro is defined as empty, this message can result.

Recommended Action: Remove the trailing semicolon.

`degenerate unsigned comparison`

An unsigned comparison is being performed against a signed value when the result is expected to be less than zero. The following program demonstrates this situation:

```
% cat x.c
#include <stdio.h>
unsigned long offset = -1;

main()
{
    if (offset < 0) {              1
        puts ("code is Ok...");
        return 0;
    } else {
        puts ("unsigned comparison failed...");
        return 1;
    }
}
% cc -g -o x x.c
% lint x.c
"x.c" line 7: warning: degenerate unsigned comparison
% ./x
unsigned comparison failed...
```

```
%
```

1 Unsigned comparisons such as this will fail if the unsigned variable contains a negative value. The resulting code may be correct, depending upon whether the programmer intended a signed comparison.

Recommended Action: You can fix the previous example in two ways:

- Add a `(long)` cast before `offset` in the `if` comparison.
- Change the declaration of `offset` from `unsigned long` to `long`.

In certain cases, it might be necessary to cast the signed value to unsigned.

`function prototype not in scope`

This error is not strictly related to function prototypes, as the message implies. Actually, this error occurs from invoking any function that has not been previously declared or defined.

Recommended Action: Add the function prototype declaration.

`null effect`

The `lint` program detected a cast or statement that does nothing. The following code segments demonstrate various coding practices that cause `lint` to generate this message:

```
scsi_slot = device->ctlr_hd->slot,unit_str;     1

#define MCLUNREF(p)             \
        (MCLMAPPED(p) && --mclrefcnt[mtocl(p)] == 0)

(void) MCLUNREF(m);                             2
```

1 Reason: `unit_str` does nothing.

2 Reason: `(void)` is unnecessary; `MCLUNREF` is a macro.

Recommended Action: Remove unnecessary casts or statements, or update macros.

`possible pointer alignment problem`

A pointer is used in a way that may cause an alignment problem. The following code segment demonstrates the type of code that causes `lint` to generate this message:

```
read(p, args, retval)
        struct proc *p;
        void *args;
        long *retval;
{
        register struct args {
                long    fdes;
                char    *cbuf;
                unsigned long  count;
        } *uap = (struct args *) args;         1
        struct uio auio;
        struct iovec aiov;
```

1 The line `*uap = (struct args *) args` causes the error to be reported. Because this construct is valid and occurs throughout the kernel source, this message is filtered out.

precision lost in field assignment

An attempt was made to assign a constant value to a bit field when the field is too small to hold the value. The following code segment demonstrates this problem:

```
% cat x.c
struct bitfield {
    unsigned int block_len : 4;
} bt;

void
test()
{
    bt.block_len = 0xff;
}
% lint -u x.c
"x.c", line 8: warning: precision lost in field assignment
% cc -c -o x x.c
%
```

This code compiles without error. However, because the bit field may be too small to hold the constant, the results may not be what the programmer intended and a run-time error may occur.

Recommended Action: Change the bit field size or assign a different constant value.

unsigned comparison with 0

An unsigned comparison is being performed against zero when the result is expected to be equal to or greater than zero.

The following program demonstrates this problem:

```
% cat z.c
#include <stdio.h>
unsigned offset = -1;
```

```
main()
{
    if (offset > 0) {              1
        puts("unsigned comparison with 0 Failed");
        return 1;
    } else {
        puts("unsigned comparison with 0 is Ok");
        return 0;
    }
}
% cc -o z z.c
% lint z.c
"z.c", line 7: warning: unsigned comparison with 0?
% ./z
unsigned comparison with 0 Failed
%
```

1 Unsigned comparisons such as this will fail if the unsigned variable contains a negative value. The resulting code may not be correct, depending on whether the programmer intended a signed comparison.

Recommended Action: You can fix the previous example in two ways:

- Add an `(int)` cast before `offset` in the `if` comparison.
- Change the declaration of `offset` from `unsigned` to `int`.

6.12 Using Warning Class Options to Suppress lint Messages

Several `lint` warning classes have been added to the `lint` program to allow the suppression of messages associated with constants used in conditionals, portability, and prototype checks. By using the warning class option to the `lint` command, you can suppress messages in any of the warning classes.

The warning class option has the following format:

-w*class* [*class*...]

All warning classes are active by default, but may be individually deactivated by including the appropriate option as part of the *`class`* argument. Table 6–1 lists the individual options.

Note

Several `lint` messages depend on more than one warning class. Therefore, you may need to specify several warning classes for the message to be suppressed. Notes in Table 6–1 indicate which messages can be suppressed only by specifying multiple warning classes.

For example, because `lint` messages related to constants in conditional expressions do not necessarily indicate a coding problem (as described in Section 6.11), you may decide to use the `-wC` option to suppress them.

The `-wC` option suppresses the following messages:

- `constant argument to NOT`
- `constant in conditional context`

Because many of the messages associated with portability checks are related to non-ANSI compilers and limit restrictions that do not exist in the C compiler for Tru64 UNIX, you can use the `-wp` option to suppress them. The `-wp` option suppresses the following messages:

- `ambiguous assignment for non-ansi compilers`
- `illegal cast in a constant expression`
- `long in case or switch statement may be truncated in non-ansi compilers`
- `nonportable character comparison`
- `possible pointer alignment problem, op %s`
- `precision lost in assignment to (sign-extended?) field`
- `precision lost in field assignment`
- `too many characters in character constant`

Although the use of function prototypes is a recommended coding practice (as described in Section 6.13), many programs do not include them. You can use the `-wP` option to suppress prototype checks. The `-wP` option suppresses the following messages:

- `function prototype not in scope`
- `mismatched type in function argument`
- `mix of old and new style function declaration`
- `old style argument declaration`
- `use of old-style function definition in presence of prototype`

Table 6–1: lint Warning Classes

Warning Class	Description of Class
a	Non-ANSI features. Suppresses: • Partially elided initialization[a] • Static function %s not defined or used[a]
c	Comparisons with unsigned values. Suppresses: • Comparison of unsigned with negative constant • Degenerate unsigned comparison • Possible unsigned comparison with 0
d	Declaration consistency. Suppresses: • External symbol type clash for %s • Illegal member use: perhaps %s.%s[b] • Incomplete type for %s has already been completed • Redeclaration of %s • Struct/union %s never defined[b] • %s redefinition hides earlier one[a b]
h	Heuristic complaints. Suppresses: • Constant argument to NOT[c] • Constant in conditional context[c] • Enumeration type clash, op %s • Illegal member use: perhaps %s.%s[d] • Null effect[e] • Possible pointer alignment problem, op %s[f] • Precedence confusion possible: parenthesize![g] • Struct/union %s never defined[d] • %s redefinition hides earlier one[d]
k	K&R type code expected. Suppresses: • Argument %s is unused in function %s[h] • Function prototype not in scope[h] • Partially elided initialization[h] • Static function %s is not defined or used[h] • %s may be used before set[b d] • %s redefinition hides earlier one[b d] • %s set but not used in function %s [h]

Table 6–1: lint Warning Classes (cont.)

Warning Class	Description of Class
l	Assign long values to non-long variables. Suppresses: • Conversion from long may lose accuracy • Conversion to long may sign-extend incorrectly
n	Null-effect code. Suppresses: • Null effect [b]
o	Unknown order of evaluation. Suppresses: • Precedence confusion possible: parenthesize! [b] • %s evaluation order undefined
p	Various portability concerns. Suppresses: • Ambiguous assignment for non-ANSI compilers • Illegal cast in a constant expression • Long in case or switch statement may be truncated in non-ANSI compilers • Nonportable character comparison • Possible pointer alignment problem, op %s [b] • Precision lost in assignment to (possibly) sign-extended field • Precision lost in field assignment • Too many characters in character constant
r	Return statement consistency. Suppresses: • Function %s has return(e); and return; • Function %s must return a value • main() returns random value to invocation environment
S	Storage capacity checks. Suppresses: • Array not large enough to store terminating null • Constant value (0x%x) exceeds (0x%x)
u	Proper usage of variables and functions. Suppresses: • Argument %s unused in function %s[a] • Static function %s not defined or used[a] • %s set but not used in function %s[a] • %s unused in function %s[h]
A	Activate all warnings. Default option in `lint` script. Specifying another A class toggles the setting of all classes.

Table 6–1: lint Warning Classes (cont.)

Warning Class	Description of Class
C	Constants occurring in conditionals. Suppresses: • Constant argument to NOT[b] • Constant in conditional context[b]
D	External declarations are never used. Suppresses: • Static %s %s unused
O	Obsolete features. Suppresses: • Storage class not the first type specifier
P	Prototype checks. Suppresses: • Function prototype not in scope[a] • Mismatched type in function argument • Mix of old- and new-style function declaration • Old-style argument declaration[a] • Use of old-style function definition in presence of prototype
R	Detection of unreachable code. Suppresses: • Statement not reached

[a] You can also suppress this message by deactiviating the k warning class.
[b] You must also deactivate the h warning class to suppress this message.
[c] You must also deactivate the C warning class to suppress this message.
[d] You must also deactivate the d warning class to suppress this message.
[e] You must also deactivate the n warning class to suppress this message.
[f] You must also deactivate the p warning class to suppress this message.
[g] You must also deactivate the o warning class to suppress this message.
[h] Other flags may also suppress these messages.

6.13 Generating Function Prototypes for Compile-Time Detection of Syntax Errors

In addition to correcting the various errors reported by the `lint` program, we recommend adding function prototypes to your program for both external and static functions. These declarations provide the compiler with information it needs to check arguments and return values.

The `cc` compiler provides an option that automatically generates prototype declarations. By specifying the `-proto[is]` option for a compilation, you create an output file (with the same name as the input file but with a `.H` extension) that contains the function prototypes. The `i` option includes identifiers in the prototype, and the `s` option generates prototypes for static functions as well.

You can copy the function prototypes from a .H file and place them in the appropriate locations in the source and include files.

7

Debugging Programs with Third Degree

The Third Degree tool checks for leaking heap memory, referencing invalid addresses and reading uninitialized memory in C and C++ programs. Programs must first be compiled with either the `-g` or `-g`*n* option, where *n* is greater than 0. Third Degree also helps you determine the allocation habits of your program by listing heap objects and finding wasted memory. It accomplishes this by instrumenting executable objects with extra code that automatically monitors memory management services and load/store instructions at run time. The requested reports are written to one or more log files that can optionally be displayed, or associated with source code by using the `xemacs`(1) editor.

By default, Third Degree checks only for memory leaks, resulting in fast instrumentation and run-time analysis. The other more expensive and intrusive checks are selected with options on the command line. See `third`(1) for more information.

You can use Third Degree for the following types of applications:

- Applications that allocate memory by using the `malloc`, `calloc`, `realloc`, `valloc`, `alloca`, and `sbrk` functions and the C++ `new` function. You can also use Third Degree to instrument programs using other memory allocators, such as the `mmap` function, but it will not check accesses to the memory obtained in this manner. If your application uses `mmap`, see the description of the `-mapbase` option in `third`(1).

  Third Degree detects and forbids calls to the `brk` function. Furthermore, if your program allocates memory by partitioning large blocks that it obtained by using the `sbrk` function, Third Degree may not be able to precisely identify memory blocks in which errors occur.

- Applications that call `fork`(2). You must specify the `-fork` option with the `third`(1) command.

- Applications that use the Tru64 UNIX implementation of POSIX threads (`pthread`(3)). You must specify the `-pthread` option with the `third`(1) command. In `pthread` programs, Third Degree does not check system-library routines (for example, `libc` and `libpthread`) for access to invalid addresses or uninitialized variables; therefore, `strcpy` and other such routines will not be checked.

- Applications that use 31–bit heap addresses.

7.1 Running Third Degree on an Application

To invoke Third Degree, use the third(1) command as follows:

third [*option...*] *app* [*argument...*]

In this command synopsis, *option* selects one or more options beyond the default nonthreaded leak checking, *app* is the name of the application, and *argument* represents one or more optional arguments that are passed to the application if you want to run the instrumented program immediately. (Use the -run option if *app* needs no arguments.)

The instrumented program, named *app*.third (see third(1)), differs from the original as follows:

- The code is larger and runs more slowly because of the additional instrumentation code that is inserted. The amount of overhead depends on the number and nature of the specified options.
- To detect errant use of uninitialized data, Third Degree initializes all otherwise uninitialized data to a special pattern (0xfff8a5a5, or as specified in the -uninit option). This can cause the instrumented program to behave differently, behave incorrectly, or crash (particularly if this special pattern is used as a pointer). All of these behaviors indicate a bug in the program. You can take advantage of this by running regression tests on the instrumented program, and you can investigate problems using the third(1) command's -g option and running a debugger. Third Degree poisons memory in this way only if the -uninit option is specified (for example, -uninit heap+stack). Otherwise, most instrumented programs run just like the original.
- Each allocated heap memory object is larger because Third Degree pads it to allow boundary checking. You can adjust the amount of padding by specifying the -pad option.
- When memory is deallocated with free or delete, it is held back from the free pool to help detect invalid access. Adjust the holding queue size with the -free option.

Third Degree writes error messages in a format similar to that used by the C compiler. It writes them to a log file named *app*.3log, by default. You can use emacs to automatically point to each error in sequence. In emacs, use Esc/X compile, replace the default make command with a command such as cat *app*.3log, and step through the errors as if they were compilation errors, with Ctrl/X` .

You can change the name used for the log file by specifying one of the following options:

–pids

Includes the process identification number (PID) in the log file name.

–dirname *directory-name*

Specifies the directory path in which Third Degree creates its log file.

-fork

Includes the PID in the name of the log file for each forked process.

Depending on the option supplied, the log file's name will be as follows:

Option	Filename	Use
None or –fork *parent*	app.3log	Default
–pids or -fork *child*	app.12345.3log	Include PID
–dirname /tmp	/tmp/app.3log	Set directory
–dirname /tmp -pids	/tmp/app.12345.3log	Set directory and PID

Errors in signal handlers may be reported in an additional .sig.3log option.

7.1.1 Using Third Degree with Shared Libraries

Errors in an application, such as passing too small a buffer to the strcpy function, are often caught in library routines. Third Degree supports the instrumentation of shared libraries; it instruments programs linked with either the –non_shared or the –call_shared option.

The following options let you determine which shared libraries are instrumented by Third Degree:

–all

Instruments all shared libraries that were linked with the call-shared executable.

–excobj *objname*

Excludes the named shared library from instrumentation. You can use the –excobj option more than once to specify several shared libraries.

–incobj *objname*

Instruments the named shared library. You can use the –incobj option more than once to specify several shared libraries, including those loaded using dlopen().

`-L`*`directory`*

Tells Third Degree where to find the program's shared libraries if they are not in the standard places known to the linker and loader.

When Third Degree finishes instrumenting the application, the current directory contains an instrumented version of each specified shared library, and at least minimally instrumented versions of `libc.so`, `libcxx.so`, and `libpthread.so`, as appropriate. The instrumented application needs to use these versions of the libraries. Define the `LD_LIBRARY_PATH` environment variable to tell the instrumented application where the instrumented shared libraries reside. The `third`(1) command will do this automatically if you specify the `-run` option or you specify arguments to the application (which also cause the instrumented program to be executed).

By default, Third Degree does not fully instrument any of the shared libraries used by the application, though it does have to minimally instrument `libc.so`, `libcxx.so`, and `libpthread.so` when used. This makes the instrumentation operation much faster and causes the instrumented application to run faster as well. Third Degree detects and reports errors in the instrumented portion normally, but it does not detect errors in the uninstrumented libraries. If your partially instrumented application crashes or malfunctions and you have fixed all of the errors reported by Third Degree, reinstrument the application and all of its shared libraries and run the new instrumented version, or use Third Degree's `-g` option to investigate the problem in a debugger.

Third Degree needs to instrument a shared library (but only minimally, by default) to generate error reports that include stack traces through its procedures. Also, a debuggable procedure (compiled with the `-g` option, for example) must appear within the first few stack frames nearest the error. This avoids printing spurious errors that the highly optimized and assembly code in system libraries can generate. Use the `-hide` option to override this feature.

For pthread programs, Third Degree does not check some system shared libraries (including `libc`) for errors, because doing so would not be thread safe.

7.2 Debugging Example

Assume that you must debug the small application represented by the following source code (`ex.c`):

```
#include <assert.h>

int GetValue() {
    int q;
    int *r=&q;
    return q;           /* q is uninitialized */
}

long* GetArray(int n) {
    long* t = (long*) malloc(n * sizeof(long));
    t[0] = GetValue();
    t[0] = t[1]+1;      /* t[1] is uninitialized */
    t[1] = -1;
    t[n] = n;           /* array bounds error*/
    if (n<10) free(t);  /* may be a leak */
    return t;
}

main() {
    long* t = GetArray(20);
    t = GetArray(4);
    free(t);            /* already freed */
    exit(0);
}
```

The following sections explain how to use Third Degree to debug this sample application.

7.2.1 Customizing Third Degree

Command-line options are used to turn on and off various capabilities of Third Degree.

If you do not specify any options, Third Degree instruments the program as follows but does not run the instrumented program or display the resulting .3log file(s):

- Detect leaks at program exit.
- Do not check for memory errors (invalid addresses or uninitialized values).
- Do not analyze the heap-usage history.

You can run the instrumented application with a command such as ./*app*.third *arg1 arg2* after setting the LD_LIBRARY_PATH environment variable. Alternatively, you can append the application arguments to the third(1) command line or specify the -run or -display options. You can view the resulting .3log file manually or by specifying the -display option.

To add checks for memory errors, specify the -invalid option and/or the -uninit option.

You can abbreviate the -invalid option, like all third(1) options, to three letters (-inv). It tells Third Degree to check that all significant load and store instructions are accessing valid memory addresses that application code should. This option carries a noticeable performance overhead, but it has little effect on the run-time environment.

The -uninit option takes a "+"-separated list of keyword arguments. This is usually heap+stack (or h+s), which asks that both heap memory and stack memory be checked for all significant load instructions. Checking involves prefilling all stack frames and heap objects allocated with malloc, and so on (but not calloc), with the unusual pattern 0xfff8a5a5, and reporting any load instruction that reads such a value out of memory. That is, the selected memory is poisoned, much as by the cc -trapuv option, to highlight code that reads uninitialized data areas. If the offending code was selected for instrumentation, Third Degree will report each case (once only) in the .3log file. However, whether or not the code was instrumented, the code will load and process the poison pattern instead of the value that the original program would have loaded. This may cause the program to malfunction or crash, because the pattern is not a valid pointer, character, or floating-point number, and it is a negative integer. Such behavior is a sign of a bug in the program.

You can identify malfunctions by running regression tests on the instrumented program, specifying -quiet and omitting -display if running within third(1). You can debug malfunctions or crashes by looking at the error messages in the .3log file and by running the instrumented program in a debugger such as dbx(1), or ladebug(1) for C++ and pthread applications. To use a debugger, compile with a -g option and specify -g on the third(1) command line as well.

The -uninit option can report false errors, particularly for variables, array elements, and structure members of less than 32 bits (for example, short, char, bit-field). See Section 7.6. However, using the -uninit heap+stack option can improve the accuracy of leak reports.

To add a heap-usage analysis, specify the -history option. This enables the -uninit heap option.

7.2.2 Modifying the Makefile

Add the following entry to the application's makefile:

```
ex.third: ex
        third ex
```

Build ex.third as follows:

```
> make ex.third
third ex
```

Now run the instrumented application ex.third and check the log ex.3log. Alternatively, run it and display the .3log file immediately by adding the -display option before the program name.

7.2.3 Examining the Third Degree Log File

The ex.3log file contains several parts that are described in the following sections, assuming this command line as an example:

```
> third -invalid -uninit h+s -history -display ex
```

7.2.3.1 List of Run-Time Memory Access Errors

The types of errors that Third Degree can detect at run-time include such conditions as reading uninitialized memory, reading or writing unallocated memory, freeing invalid memory, and certain serious errors likely to cause an exception. For each error, an error entry is generated with the following items:

- A banner line with the type of error and number — The error banner line contains a three-letter abbreviation of each error (see Section 7.3 for a list of the abbreviations). If the process that caused the error is not the root process (for instance, because the application forks one or more child processes), the PID of the process that caused the error also appears in the banner line.
- An error message line formatted to look like a compiler error message — Third Degree lists the file name and line number nearest to the location where the error occurred. Usually this is the precise location where the error occurred, but if the error occurs in a library routine, it can also point to the place where the library call occurred.
- One or more stack traces — The last part of an error entry is a stack trace. The first procedure listed in the stack trace is the procedure in which the error occurred.

The following examples show entries from the log file:

- The following log entry indicates that a local variable of procedure GetValue was read before being initialized. The line number confirms that q was never given a value.

```
---------------------------------------------------- rus -- 0 --
ex.c: 6: reading uninitialized local variable q of GetValue
    GetValue                       ex, ex.c, line 6
    GetArray                       ex, ex.c, line 11
    main                           ex, ex.c, line 20
    __start                        ex
```

- In the following log entry, an error is reported at line 12:

  `t[0] = t[1]+1`

  Because the array was not initialized, the program is using the uninitialized value of `t[1]` in the addition. The memory block containing array `t` is identified by the call stack that allocated it. Stack variables are identified by name if the code was compiled with the `-g` option.

```
---------------------------------------------------- ruh -- 1 --
ex.c: 12: reading uninitialized heap at byte 8 of 160-byte block
    GetArray                       ex, ex.c, line 12
    main                           ex, ex.c, line 20
    __start                        ex

This block at address 0x14000ca20 was allocated at:
    malloc                         ex
    GetArray                       ex, ex.c, line 10
    main                           ex, ex.c, line 20
    __start                        ex
```

- The following log entry indicates that the program has written to the memory location one position past the end of the array, potentially overwriting important data or even Third Degree internal data structures. Keep in mind that certain errors reported later could be a consequence of this error:

```
---------------------------------------------------- wih -- 2 --
ex.c: 14: writing invalid heap 1 byte beyond 160-byte block
    GetArray                       ex, ex.c, line 14
    main                           ex, ex.c, line 20
    __start                        ex

This block at address 0x14000ca20 was allocated at:
    malloc                         ex
    GetArray                       ex, ex.c, line 10
    main                           ex, ex.c, line 20
    __start                        ex
```

- The following log entry indicates that an error occurred while freeing memory that was previously freed. For errors involving calls to the `free` function, Third Degree usually gives three call stacks:

  – The call stack where the error occurred

  – The call stack where the object was allocated

  – The call stack where the object was freed

  Upon examining the program, it is clear that the second call to `GetArray` (line 20) frees the object (line 14), and that another attempt to free the same object occurs at line 21:

```
---------------------------------------------------- fof -- 3 --
ex.c: 22: freeing already freed heap at byte 0 of 32-byte block
    free                           ex
    main                           ex, ex.c, line 22
    __start                        ex
This block at address 0x14000d1a0 was allocated at:
    malloc                         ex
    GetArray                       ex, ex.c, line 10
    main                           ex, ex.c, line 21
    __start                        ex

This block was freed at:
    free                           ex
    GetArray                       ex, ex.c, line 15
    main                           ex, ex.c, line 21
    __start                        ex
```

See Section 7.3 for more information.

7.2.3.2 Memory Leaks

The following excerpt shows the report generated when leak detection on program exit, the default, is selected. The report shows a list of memory leaks sorted by importance and by call stack.

```
------------------------------------------------------------------------
------------------------------------------------------------------------
New blocks in heap after program exit

Leaks - blocks not yet deallocated but apparently not in use:
 * A leak is not referenced by static memory, active stack frames,
   or unleaked blocks, though it may be referenced by other leaks.
 * A leak "not referenced by other leaks" may be the root of a leaked tree.
 * A block referenced only by registers, unseen thread stacks, mapped memory,
   or uninstrumented library data is falsely reported as a leak. Instrumenting
   shared libraries, if any, may reduce the number of such cases.
 * Any new leak lost its last reference since the previous heap report, if any.

A total of 160 bytes in 1 leak were found:

160 bytes in 1 leak (including 1 not referenced by other leaks) created at:
    malloc                         ex
    GetArray                       ex, ex.c, line 10
    main                           ex, ex.c, line 20
    __start                        ex

Objects - blocks not yet deallocated and apparently still in use:
 * An object is referenced by static memory, active stack, or other objects.
 * A leaked block may be falsely reported as an object if a pointer to it
   remains when a new stack frame or heap block reuses the pointer’s memory.
   Using the option to report uninitialized stack and heap may avoid such cases.
 * Any new object was allocated since the previous heap report, if any.

A total of 0 bytes in 0 objects were found:
```

Upon examining the source, it is clear that the first call to `GetArray` did not free the memory block, nor was it freed anywhere else in the program. Moreover, no pointer to this block exists in any other heap block, so it qualifies as “`not referenced by other leaks`”. The distinction is often useful to find the real culprit for large memory leaks.

Consider a large tree structure and assume that the pointer to the root has been erased. Every block in the structure is a leak, but losing the pointer to the root is the real cause of the leak. Because all blocks but the root still have pointers to them, albeit only from other leaks, only the root will be specially qualified, and therefore the likely cause of the memory loss.

See Section 7.4 for more information.

7.2.3.3 Heap History

When heap history is enabled, Third Degree collects information about dynamically allocated memory. It collects this information for every block that is freed by the application and for every block that still exists (including memory leaks) at the end of the program's execution. The following excerpt shows a heap allocation history report:

```
------------------------------------------------------------------------
------------------------------------------------------------------------
                  Heap Allocation History for parent process


Legend for object contents:
        There is one character for each 32-bit word of contents.
        There are 64 characters, representing 256 bytes of memory per line.
        '.' : word never written in any object.
        'z' : zero in every object.
        'i' : a non-zero non-pointer value in at least one object.
        'pp': a valid pointer or zero in every object.
        'ss': a valid pointer or zero in some but not all objects.

192 bytes in 2 objects were allocated during program execution:

--------------------------------------------------------------------------
160 bytes allocated (8% written) in 1 objects created at:
    malloc                         ex
    GetArray                       ex, ex.c, line 10
    main                           ex, ex.c, line 20
    __start                        ex

Contents:
     0: i.ii....................................

--------------------------------------------------------------------------
32 bytes allocated (38% written) in 1 objects created at:
    malloc                         ex
    GetArray                       ex, ex.c, line 10
    main                           ex, ex.c, line 21
    __start                        ex

Contents:
     0: i.ii....
```

The sample program allocated two objects for a total of 192 bytes (8*(20+4)). Because each object was allocated from a different call stack, there are two entries in the history. Only the first few bytes of each array were set to a valid value, resulting in the written ratios shown.

If the sample program was a real application, the fact that so little of the dynamic memory was ever initialized is a warning that it was probably using memory ineffectively.

See Section 7.4.4 for more information.

7.2.3.4 Memory Layout

The memory layout section of the report summarizes the memory used by the program by size and address range. The following excerpt shows a memory layout section:

```
-----------------------------------------------------------------
-----------------------------------------------------------------
                memory layout at program exit
                  heap      40960 bytes [0x14000c000-0x140016000]
                 stack       2720 bytes [0x11ffff560-0x120000000]
               ex data      48528 bytes [0x140000000-0x14000bd90]
               ex text    1179648 bytes [0x120000000-0x120110000]
```

The heap size and address range indicated reflect the value returned by `sbrk(0)`, (the heap break) at program exit. Therefore, the size is the total amount of heap space that has been allotted to the process. Third Degree does not support the use of the `malloc` variables that would alter this interpretation of `sbrk(0)`.

The stack size and address range reflect the lowest address reached by the main thread's stack pointer during execution of the program. That is, Third Degree keeps track of it through each instrumented procedure call. For this value to reflect the maximum stack size, all shared libraries need to have been instrumented (for example, using the `third`(1) command's `-all` option for a nonthreaded program and `-incobj` options for libraries loaded with `dlopen(3)`). The stacks of threads (created using `pthread_create`) are not included.

The data and text sizes and address ranges show where the static portions of the executable and each shared library were loaded.

7.3 Interpreting Third Degree Error Messages

Third Degree reports both fatal errors and memory access errors. Fatal errors include the following:

- Bad parameter

  For example, `malloc(-10)`.

- Failed allocator

  For example, `malloc` returned a zero, indicating that no memory is available.

- Call to the `brk` function with a nonzero argument

  Third Degree does not allow you to call `brk` with a nonzero argument.

- Memory allocation not permitted in signal handler.

A fatal error causes the instrumented application to crash after flushing the log file. If the application crashes, first check the log file and then rerun it under a debugger, having specified `-g` on the `third`(1) command line.

Memory errors include the following (as represented by a three-letter abbreviation):

Name	Error
`ror`	Reading out of range: not heap, stack, or static (for example, NULL)
`ris`	Reading invalid data in stack: probably an array bound error
`rus`	Reading an uninitialized (but valid) location in stack
`rih`	Reading invalid data in heap: probably an array bound error
`ruh`	Reading an uninitialized (but valid) location in heap
`wor`	Writing out of range: neither in heap, stack, or static area
`wis`	Writing invalid data in stack: probably an array bound error
`wih`	Writing invalid data in heap: probably an array bound error
`for`	Freeing out of range: neither in heap or stack
`fis`	Freeing an address in the stack
`fih`	Freeing an invalid address in the heap: no valid object there
`fof`	Freeing an already freed object
`fon`	Freeing a null pointer (really just a warning)
`mrn`	`malloc` returned null

You can suppress the reporting of specific memory errors by specifying one or more `-ignore` options. This is often useful when the errors occur within library functions for which you do not have the source. Third Degree allows you to suppress specific memory errors in individual procedures and files, and at particular line numbers. See `third`(1) for more details.

Alternatively, do not select the library for checking, by specifying `-excobj` or omitting the `-all` or `-incobj` option.

7.3.1 Fixing Errors and Retrying an Application

If Third Degree reports many write errors from your instrumented program, fix the first few errors and then reinstrument the program. Not only can write errors compound, but they can also corrupt Third Degree's internal data structures.

7.3.2 Detecting Uninitialized Values

Third Degree's technique for detecting the use of uninitialized values can cause programs that have worked to fail when instrumented. For example, if a program depends on the fact that the first call to the `malloc` function returns a block initialized to zero, the instrumented version of the program will fail because Third Degree poisons all blocks with a nonzero value (0xfff8a5a5, by default).

When it detects a signal, perhaps caused by dereferencing or otherwise using this uninitialized value, Third Degree displays a message of the following form:

```
*** Fatal signal SIGSEGV detected.
*** This can be caused by the use of uninitialized data.
*** Please check all errors reported in app.3log.
```

Using uninitialized data is the most likely reason for an instrumented program to crash. To determine the cause of the problem, first examine the log file for reading-uninitialized-stack and reading-uninitialized-heap errors. Very often, one of the last errors in the log file reports the cause of the problem.

If you have trouble pinpointing the source of the error, you can confirm that it is indeed due to reading uninitialized data by removing one of the `heap` and `stack` options from the `-uninit` option (or the whole option). Removing `stack` disables the poisoning of newly allocated stack memory that Third Degree normally performs on each procedure entry. Similarly, removing `heap` disables the poisoning of heap memory performed on each dynamic memory allocation. By using one or both options, you can alter the behavior of the instrumented program and may likely get it to complete successfully. This will help you determine which type of error is causing the instrumented program to crash and, as a result, help you focus on specific messages in the log file.

Alternatively, run the instrumented program in a debugger (using the `-g` option of the `third`(1) command) and remove the cause of the failure. You need not use the `-uninit` option if you just want to check for memory leaks; however, using the `-uninit` option can make the leak reports more accurate.

If your program establishes signal handlers, there is a small chance that Third Degree's changing of the default signal handler may interfere with it. Third Degree defines signal handlers only for those signals that normally cause program crashes (including `SIGILL`, `SIGTRAP`, `SIGABRT`, `SIGEMT`, SIGFPE, SIGBUS, SIGSEGV, SIGSYS, SIGXCPU, and SIGXFSZ). You can disable Third Degree's signal handling by specifying the -signals option.

7.3.3 Locating Source Files

Third Degree prefixes each error message with a file and line number in the style used by compilers. For example:

```
----------------------------------------------------- fof -- 3 --
ex.c: 21: freeing already freed heap at byte 0 of 32-byte block
    free                              malloc.c
    main                              ex.c, line 21
    __start                           crt0.s
```

Third Degree tries to point as closely as possible to the source of the error, and it usually gives the file and line number of a procedure near the top of the call stack when the error occurred, as in this example. However, Third Degree may not be able to find this source file, either because it is in a library or because it is not in the current directory. In this case, Third Degree moves down the call stack until it finds a source file to which it can point. Usually, this is the point of call of the library routine.

To tag these error messages, Third Degree must determine the location of the program's source files. If you are running Third Degree in the directory containing the source files, Third Degree will locate the source files there. If not, to add directories to Third Degree's search path, specify one or more -use options. This allows Third Degree to find the source files contained in other directories. The location of each source file is the first directory on the search path in which it is found.

7.4 Examining an Application's Heap Usage

In addition to run-time checks that ensure that only properly allocated memory is accessed and freed, Third Degree provides two ways to understand an application's heap usage:

- It can find and report memory leaks.
- It can list the contents of the heap.

By default, Third Degree checks for leaks when the program exits.

This section discusses how to use the information provided by Third Degree to analyze an application's heap usage.

7.4.1 Detecting Memory Leaks

A memory leak is an object in the heap to which no in-use pointer exists. The object can no longer be accessed and can no longer be used or freed. It is useless, will never go away, and wastes memory.

Third Degree finds memory leaks by using a simple trace-and-sweep algorithm. Starting from a set of roots (the currently active stack and the static areas), Third Degree finds pointers to objects in the heap and marks these objects as visited. It then recursively finds all potential pointers inside these objects and, finally, sweeps the heap and reports all unmarked objects. These unmarked objects are leaks.

The trace-and-sweep algorithm finds all leaks, including circular structures. This algorithm is conservative: in the absence of type information, any 64-bit pattern that is properly aligned and pointing inside a valid object in the heap is treated as a pointer. This assumption can infrequently lead to the following problems:

- Third Degree considers pointers either to the beginning or interior of an object as true pointers. Only objects with no pointers to any address they contain are considered leaks.
- If an instrumented application hides true pointers by storing them in the address space of some other process or by encoding them, Third Degree will report spurious leaks. When instrumenting such an application with Third Degree, specify the `-mask` option. This option lets you specify a mask that is applied as an AND operator against every potential pointer. For example, if you use the top three bits of pointers as flags, specify a mask of 0x1fffffffffffffff. See `third`(1).
- Third Degree can confuse any bit pattern (such as string, integer, floating-point number, and packed struct) that looks like a heap pointer with a true pointer, thereby missing a true leak.
- Third Degree does not notice pointers that optimized code stores only in registers, not in memory. As a result, it may produce false leak reports.

To maximize the accuracy of the leak reports, use the `-uninit h+s` and `-all` options. However, the `-uninit` option can cause the program to fail, and the `-all` option increases the instrumentation and run time. So, just check both the Leaks and Objects listings, and evaluate for possible program errors.

7.4.2 Reading Heap and Leak Reports

You can supply command options that tell Third Degree to generate heap and leak reports incrementally, listing only new heap objects or leaks since the last report or listing all heap objects or leaks. You can request these reports when the program terminates, or before or after every *n*th call to a user-specified function. See `third`(1) for details of the `-blocks`, `-every`, `-before`, and `-after` options. The `-blocks` option (the default) reports both the leaks and the objects in the heap, so you will never miss one in the event that it is classified as the wrong type. The `.3log` file describes the situations where incorrect classification can occur, along with ways to improve its accuracy.

You should pay closest attention to the leaks report, because Third Degree has found evidence suggesting that the reported blocks really are leaked, whereas the evidence suggests that the blocks reported as objects were not. However, if your debugging and examination of the program suggests otherwise, you can reasonably deduce that the evidence was misleading to the tool.

Third Degree lists memory objects and leaks in the report by decreasing importance, based on the number of bytes involved. It groups together objects allocated with identical call stacks. For example, if the same call sequence allocates a million one-byte objects, Third Degree reports them as a 1-MB group containing a million allocations.

To tell Third Degree when objects or leaks are the same and should be grouped in the report (or when objects or leaks are different and should not be thus grouped), specify the `-depth` option. It sets the depth of the call stack that Third Degree uses to differentiate leaks or objects. For example, if you specify a depth of 1 for objects, Third Degree groups valid objects in the heap by the function and line number that allocated them, no matter what function was the caller. Conversely, if you specify a very large depth for leaks, Third Degree groups only leaks allocated at points with identical call stacks from `main` upwards.

In most heap reports, the first few entries account for most of the storage, but there is a very long list of small entries. To limit the length of the report, you can use the `-min` option. It defines a percentage of the total memory leaked or in use by an object as a threshold. When all smaller remaining leaks or objects amount to less than this threshold, Third Degree groups them together under a single final entry.

Notes

Because the `realloc` function always allocates a new object (by involving calls to `malloc`, `copy`, and `free`), its use can make interpretation of a Third Degree report counterintuitive. An object can be listed twice under different identities.

Leaks and objects are mutually exclusive: an object must be reachable from the roots.

7.4.3 Searching for Leaks

It may not always be obvious when to search for memory leaks. By default, Third Degree checks for leaks after program exit, but this may not always be what you want.

Leak detection is best done as near as possible to the end of the program while all used data structures are still in scope. Remember, though, that the roots for leak detection are the contents of the stack and static areas. If your program terminates by returning from `main` and the only pointer to one of its data structures was kept on the stack, this pointer will not be seen as a root during the leak search, leading to false reporting of leaked memory. For example:

```
main (int argc, char* argv[]) {
    char* bytes = (char*) malloc(100);
    exit(0);
}
```

When you instrument this program, specifying `-blocks all -before exit` will cause Third Degree to not find any leaks. When the program calls the `exit` function, all of `main`'s variables are still in scope.

However, consider the following example:

```
main (int argc, char* argv[]) {
    char* bytes = (char*) malloc(100);
}
```

When you instrument this program, providing the same (or no) options, Third Degree's leak check may report a storage leak because `main` has returned by the time the check happens. Either of these two behaviors may be correct, depending on whether `bytes` was a true leak or simply a data structure still in use when `main` returned.

Rather than reading the program carefully to understand when leak detection should be performed, you can check for new leaks after a specified number of calls to the specified procedure. Use the following options to disable the default leak-checking and to request a leak before every 10,000th call to the procedure `proc_name`:

```
-blocks cancel
-blocks new -every 10000 -before proc_name
```

7.4.4 Interpreting the Heap History

When you instrument a program using the `-history` option, Third Degree generates a heap history for the program. A heap history allows you to see how the program used dynamic memory during its execution. For example, you can use this feature to eliminate unused fields in data structures or to pack active fields to use memory more efficiently. The heap history also shows memory blocks that are allocated but never used by the application.

When heap history is enabled, Third Degree collects information about each dynamically allocated object at the time it is freed by the application. When program execution completes, Third Degree assembles this information for every object that is still alive (including memory leaks). For each object, Third Degree looks at the contents of the object and categorizes each word as never written by the application, zero, a valid pointer, or some other value.

Third Degree next merges the information for each object with what it has gathered for all other objects allocated at the same call stack in the program. The result provides you with a cumulative picture of the use of all objects of a given type.

Third Degree provides a summary of all objects allocated during the life of the program and the purposes for which their contents were used. The report shows one entry per allocation point (for example, a call stack where an allocator function such as `malloc` or `new` was called). Entries are sorted by decreasing volume of allocation.

Each entry provides the following:

- Information about all objects that have been allocated
- Total number of bytes allocated
- Total number of objects that have been allocated
- Percentage of bytes of the allocated objects that have been written
- The call stack and a cumulative map of the contents of all objects allocated by that call stack

The contents part of each entry describes how the objects were used. If all allocated objects are not the same size, Third Degree considers only the minimum size common to all objects. For very large allocations, it summarizes the contents of only the beginning of the objects, by default, the first kilobyte. You can adjust the maximum size value by specifying the `-size` option.

In the contents portion of an entry, Third Degree uses one of the following characters to represent each 32-bit longword that it examines:

Character	Description
Dot (.)	Indicates a longword that was never written in any of the objects, a definite sign of wasted memory. Further analysis is generally required to see if it is simply a deficiency of a test that never used this field, if it is a padding problem solved by swapping fields or choosing better types, or if this field is obsolete.
z	Indicates a field whose value was always 0 (zero) in every object.
pp	Indicates a pointer: that is, a 64-bit quantity that was a valid pointer into the stack, the static data area, the heap (or was zero).
ss	Indicates a sometime pointer. This longword looked like a pointer in at least one of the objects, but not in all objects. It could be a pointer that is not initialized in some instances, or a union. However, it could also be the sign of a serious programming error.
i	Indicates a longword that was written with some nonzero value in at least one object and that never contained a pointer value in any object.

Even if an entry is listed as allocating 100 MB, it does not mean that at any point in time 100 MB of heap storage were used by the allocated objects. It is a cumulative figure; it indicates that this point has allocated 100 MB over the lifetime of the program. This 100 MB may have been freed, may have leaked, or may still be in the heap. The figure simply indicates that this allocator has been quite active.

Ideally, the fraction of the bytes actually written should always be close to 100 percent. If it is much lower, some of what is allocated is never used. The common reasons why a low percentage is given include the following:

- A large buffer was allocated but only a small fraction was ever used.
- Parts of every object of a given type are never used. They may be forgotten fields or padding between real fields resulting from alignment rules in C structures.
- Some objects have been allocated but never used at all. Sometimes leak detection will find these objects if their pointers are discarded. However, if they are kept on a free list, they will be found only in the heap history.

7.5 Using Third Degree on Programs with Insufficient Symbolic Information

If the executable you instrumented contains too little symbolic information for Third Degree to pinpoint some program locations, Third Degree prints messages in which procedure names or file names or line numbers are unknown. For example:

```
------------------------------------------------------ rus -- 0 --
reading uninitialized stack at byte 40 of 176 in frame of main
    proc_at_0x1200286f0                libc.so
    pc = 0x12004a268                   libc.so
    main                               app, app.c, line 16
    __start                            app
```

Third Degree tries to print the procedure name in the stack trace, but if the procedure name is missing (because this is a static procedure), Third Degree prints the program counter in the instrumented program. This information enables you to find the location with a debugger. If the program counter is unavailable, Third Degree prints the number of the unknown procedure.

Most frequently, the file name or line number is unavailable because the file was compiled with the default `-g0` option. In this case, Third Degree prints the name of the object in which the procedure was found. This object may be either the main application or a shared library.

By default, error reports are printed only if a stack frame with a source file name and a line number appears within two frames of the top of the stack. This hides spurious reports that can be caused by the highly optimized and assembly language code in the system libraries. Use the `-hide` option to hide fewer (or more) reports involving nondebuggable procedures.

If the lack of symbolic information is hampering your debugging, consider recompiling the program with more symbolic information. Recompile with the `–g` or `-g1`option and link without the `–x` option. Using `-g` will make variable names appear in reports instead of the byte offset shown in the previous example.

7.6 Validating Third Degree Error Reports

The following spurious errors can occur:

- Modifications to variables, array elements, or structure members that are less than 32 bits in size (such as `short`, `char`, bit field), as in this example:

  ```
  void Packed() {
      char c[4];
      struct { int a:6; int b:9; int c:4} x;
      c[0] = c[1] = 1;       /* rus errors here ... */
      x.a = x.c = x.e  = 3;  /* ... maybe here */
  }
  ```

  Ignore any implausible error messages, such as those reported for `strcpy`, `memcpy`, `printf`, and so on.

- Third Degree poisons newly allocated memory with a special value to detect references to uninitialized variables (see Section 7.3.2). Programs that explicitly store this special value into memory and subsequently read it may cause spurious “reading uninitialized memory” errors.

- Variable size stack frames are not supported. Ignore any messages about “invalid stack” in such cases.

If you think that you have found a false positive, you can verify it by using a debugger on the procedure in which the error was reported. All errors reported by Third Degree are detected at loads and stores in the application, and the line numbers shown in the error report match those shown in the disassembly output. Compile and instrument the program with the `-g` option before debugging.

7.7 Undetected Errors

Third Degree can fail to detect real errors, such as the following:

- Errors in operations on quantities smaller than 32 bits can go undetected (for example, `char`, `short`, and bit-field). The `-uninit repeat` option can expose such errors by checking more load and store operations, which Third Degree usually considers too low a risk to check.
- Third Degree cannot detect a chance access of the wrong object in the heap. It can only detect memory accesses from objects. For example, Third Degree cannot determine that `a[last+100]` is the same address as `b[0]`. You can reduce the chances of this happening by altering the amount of padding added to objects. To do this, specify the `-pad` option.
- Third Degree may not be able to detect if the application walks past the end of an array unless it also walks past the end of the array’s stack frame or its heap object. Because Third Degree brackets objects in the heap by guard words, it will miss small array bounds errors. (Guard words are spare memory added at the ends of valid memory blocks to detect overshoots.) In the stack, adjacent memory is likely to contain local variables, and Third Degree may fail to detect larger bounds errors. For example, issuing a `sprintf` operation to a local buffer that is much too small may be detected, but if the array bounds are only exceeded by a few words and enough local variables surround the array, the error can go undetected. Use the `cc` command’s `-check_bounds` option to detect array bounds violations more precisely.
- Hiding pointers by encoding them or by keeping pointers only to the inside of a heap object will degrade the effectiveness of Third Degree’s leak detection.
- Third Degree may detect more uninitialized variables if compiler optimization is disabled (that is, with the `-O0` and `-inline none` options).

- At times, some leaks may not be reported, because old pointers were found in memory. Selecting checks for uninitialized heap memory (`-uninit heap`) may reduce this problem.
- Any degree of optimization will skew leak-reporting results, because instructions that the compiler considers nonessential may be optimized away.

8 Profiling Programs to Improve Performance

Profiling is a method of identifying sections of code that consume large portions of execution time. In a typical program, most execution time is spent in relatively few sections of code. To improve performance, the greatest gains result from improving coding efficiency in time-intensive sections.

Tru64 UNIX supports four approaches to performance improvement:

- Automatic and profile-directed optimizations. (See Section 10.1, which covers all such optimization techniques).
- Manual design and code optimizations (Section 8.3)
- Minimizing system-resource usage (Section 8.4)
- Verifying the significance of test cases (Section 8.5)

One approach might be enough, but more might be beneficial if no single approach addresses all aspects of a program's performance. Automatic and profile-directed optimizations provide the simplest means of performance improvement. This chapter describes the last three (manual) performance improvement techniques and the tools provided by Tru64 UNIX to support them. In addition, the following topics are covered in this chapter:

- The source code for a sample program (`sample`) that is used in examples throughout this chapter (Section 8.1)
- Compilation options for profiling (Section 8.2)
- Selecting profiling information to display (Section 8.6)
- Merging profile data files (Section 8.7)
- Profiling multithreaded applications (Section 8.8)
- Using monitor routines to control profiling (Section 8.9)

For more information, see the following reference pages:

- Profiling: `cc`(1), `hiprof`(1), `pixie`(1), `third`(1), `uprofile`(1), `prof`(1), `gprof`(1)
- System monitoring: `ps`(1), `swapon`(8), `vmstat`(1)
- Performance Manager, available from the Tru64 UNIX Associated Products Volume 1 installation media: `pmgr`(8X)

- Graphical tools, available from the Graphical Program Analysis subset of the Tru64 UNIX Associated Products Volume 1 installation media, or as part of the Compaq Enterprise Toolkit for Windows/NT desktops with the Microsoft VisualStudio97: `dxheap`(1), `dxprof`(1), `mview`(1), `pview`(1)
- Visual Threads, available from the Tru64 UNIX Associated Products Volume 1 installation media: `dxthreads`(1). Use Visual Threads to analyze a multithreaded appplication for potential logic and performance problems.
- The *System Configuration and Tuning* manual.

8.1 Profiling Sample Program

The examples in this chapter refer to the program `sample`, whose source code is contained in the files `profsample.c` (main program), `add_vector.c`, `mul_by_scalar.c`, `print_it.c`, and `profsample.h`. These files are shown in Example 8–1.

Example 8–1: Profiling Sample Program

```
****************** profsample.c *************
#include <math.h>
#include <stdio.h>
#include "profsample.h"

static void mul_by_pi(double ary[])
{
    mul_by_scalar(ary, LEN/2, 3.14159);
}

void main()
{
    double ary1[LEN];
    double *ary2;
    double sum = 0.0;
    int i;

    ary2 = malloc(LEN * sizeof(double));
    for (i=0; i<LEN; i++) {
        ary1[i] = 0.0;
        ary2[i] = sqrt((double)i);
    }
    mul_by_pi(ary1);
    mul_by_scalar(ary2, LEN, 2.71828);
    for (i=0; i<100; i++)
        add_vector(ary1, ary2, LEN);
    for (i=0; i<100; i++)
 sum += ary1[i];
    if (sum < 0.0)
```

Example 8–1: Profiling Sample Program (cont.)

```
 print_it(0.0);
    else
 print_it(sum);
}
****************** profsample.h: ********************

void mul_by_scalar(double ary[], int len, double num);
void add_vector(double arya[], double aryb[], int len);
void print_it(double value);
#define LEN 100000

***************** add_vector.c: ********************

#include "profsample.h"

void add_vector(double arya[], double aryb[], int len)
{
    int i;
    for (i=0; i<LEN; i++) {
        arya[i] += aryb[i];
    }
}
**************** mul_by_scalar.c: ******************

#include "profsample.h"

void mul_by_scalar(double ary[], int len, double num)
{
    int i;
    for (i=0; i<LEN; i++) {
        ary[i] *= num;
    }
}
**************** print_it.c: **********************

#include <stdio.h>
#include "profsample.h"

void print_it(double value)
{
    printf("Value = %f\n", value);
}
```

8.2 Compilation Options for Profiling

When using debug and optimization options with the `cc` command, note the following considerations. They apply to all the profiling tools discussed in this chapter unless stated otherwise.

- The `-g1` option provides the minimum debug information needed (that is, line numbers and procedure names) and is sufficient for all profilers. The `cc` command default, `-g0`, is tolerated but provides no names for local (for example, static) procedures. The `-g2` and higher options provide less than optimal code as well as unneeded information.
- When doing manual optimization, note that none of the profiling tools show inlined procedures separately by their own names. Profiling statistics for an inlined procedure are included in the statistics for the calling procedure. For example, if `proc1` calls `proc2` (which is inlined), the statistics for `proc1` will include the time spent in `proc2`. Therefore, to provide some optimization but with minimal inlining when profiling, use the `-O2` (or `-O`) option. In some cases, you may need to specify the `-inline none` option to eliminate all inlining. This restriction does not apply to automatic optimization as described in Section 10.1.

8.3 Manual Design and Code Optimizations

The following sections discuss the techniques and tools used for manual design and code optimizations.

8.3.1 Techniques

The effectiveness of the automatic optimizations described in Section 10.1 is limited by the efficiency of the algorithms that the program uses. You can further improve a program's performance by manually optimizing its algorithms and data structures. Such optimizations may include reducing complexity from N-squared to log-N, avoiding copying of data, and reducing the amount of data used. It may also extend to tuning the algorithm to the architecture of the particular machine it will be run on — for example, processing large arrays in small blocks such that each block remains in the data cache for all processing, instead of the whole array being read into the cache for each processing phase.

Tru64 UNIX supports manual optimization with its profiling tools, which identify the parts of the application that impose the highest CPU load — CPU cycles, cache misses, and so on. By evaluating various profiles of a program, you can identify which parts of the program use the most CPU resources, and you can then redesign or recode algorithms in those parts to use less resources. The profiles also make this exercise more cost-effective by helping you to focus on the most demanding code rather than on the least demanding.

8.3.2 Tools and Examples

The following sections discuss the tools used for CPU-time profiling with call graph and for CPU-time/event profiling with source lines and instructions.

8.3.2.1 CPU-Time Profiling with Call Graph

A call-graph profile shows how much CPU time is used by each procedure, and how much is used by all the other procedures that it calls. This profile can show which phases or subsystems in a program spend most of the total CPU time, which can help in gaining a general understanding of the program's performance. This section describes two tools that provide this information:

- The `hiprof` profiler (Section 8.3.2.1.1)
- The `cc` command's `-pg` option (Section 8.3.2.1.2)

Both tools are used with the `gprof` tool, implicitly or explicitly, to format and display the profile.

The optional `dxprof` command provides a graphical display of CPU-time call-graph profiles.

8.3.2.1.1 Using the hiprof Profiler

The `hiprof` profiler (see `hiprof`(1)) instruments the program and generates a call graph while the instrumented program executes. This profiler does not require that the program be compiled in any particular way except as indicated in Section 8.2. The `hiprof` command can generate a call-graph profile for shared libraries and for program code, with moderate optimization and minimal debug information. For example:

```
% cc -o libsample.so -shared -g1 -O2 add_vector.c mul_by_scalar.c print_it.c[1]
% cc -o sample -g1 -O2 profsample.c -L. -lsample -lm[2]
% hiprof -numbers -L. -inc libsample.so sample[3]
```

[1] A shared library, `libsample.so`, is created from three source modules, with debug information and optimization as indicated in Section 8.2.

[2] The source module `profsample` is compiled and linked against the shared library `libsample.so` (located in the current directory) to produce the executable `sample`.

[3] The `-inc[obj]` option tells `hiprof` to profile `libsample.so` in addition to the executable (`sample`). The `hiprof` command creates an instrumented version of the program (`sample.hiprof`). Because at least one `gprof` option (`-numbers`) is specified, `hiprof` then

automatically runs that instrumented program to create a profile data file (sample.hiout) and then runs gprof to display the profile. The -numbers option prints each procedure's starting line number, source-file name, and library name. This helps identify any multiple static procedures with the same name.

The resulting sample profile is shown in Example 8–2. The call-graph profile estimates the total cost of calling a procedure, including other procedures that it calls. The estimation assumes that each call to a given procedure takes the same amount of time; this may not be true in many cases, but it is always accurate when there is only one caller.

By default, hiprof uses a low-frequency sampling technique. This can profile all the code executed by the program, including all selected libraries. It also provides a call-graph profile of all selected procedures (except those in the threads-related system libraries) and detailed profiles at the level of source lines or machine instructions (if selected).

Example 8–2: Sample hiprof Default Profile Using gprof

```
% cc -o libsample.so -shared -g1 -O2 add_vector.c mul_by_scalar.c print_it.c
add_vector.c:
mul_by_scalar.c:
print_it.c:

% cc -o sample -g1 -O2 profsample.c -L. -lsample -lm

% hiprof -numbers -L. -inc libsample.so sample
hiprof: info: instrumenting sample ...

hiprof: info: the LD_LIBRARY_PATH environment variable is not defined
hiprof: info: setting LD_LIBRARY_PATH=.:. 1
hiprof: info: running instrumented program sample.hiprof ...

Value = 179804.149985

hiprof: info: instrumented program exited with status 0
hiprof: info: displaying profile for sample ...

gprof -b -scaled -all -numbers -L. sample.hiprof ./sample.hiout 2

*********************** call-graph profile ******************* 3

granularity: samples per 4 bytes; units: seconds*1e-3; total: 323e-3 seconds

                                    called /     total    parents
index  %total   self  descendents   called +      self  name      index
                                    called /     total    children

                     2       321        1 /         1    __start [2]
[1]    100.0         2       321        1              main [1] 4
                   318         0      100 /       100    add_vector [3] 5
                     3         0        2 /         2    mul_by_scalar [4]
                     0         0        1 /         1    print_it [5]

------------------------------------------------------

                   318         0      100 /       100    main [1] 6
```

Example 8–2: Sample hiprof Default Profile Using gprof (cont.)

```
[3]     98.5       318         0       100             add_vector [3]

------------------------------------------------------

                     3         0         2 /         2    main [1]
[4]      0.9         3         0         2             mul_by_scalar [4]

------------------------------------------------------

                     0         0         1 /         1    main [1]
[5]      0.0         0         0         1             print_it [5]

------------------------------------------------------

*********************** timing profile section ***************

granularity: samples per 4 bytes; units: seconds*1e-3; total: 323e-3 seconds

  %   cumulative     self           self/call total/call
total     units     units     calls   seconds   seconds  name
 98.5       318       318       100   3184e-6   3184e-6 add_vector [3]
  0.9       321         3         2   1465e-6   1465e-6 mul_by_scalar [4]
  0.6       323         2         1   1953e-6 323242e-6 main [1]
  0.0       323         0         1         0         0 print_it [5]

*********************** index section ************************
Index by function name - 4 entries

   [3] add_vector         :"add_vector.c":1
   [1] main               :"profsample.c":12
   [4] mul_by_scalar      :"mul_by_scalar.c":1
   [5] print_it           :"print_it.c":4
```

1 The `LD_LIBRARY_PATH` environment variable is automatically set to point to the working directory, where the instrumented shared library `libsample` is located (see Section 8.6.4.1).

2 The automatically generated `gprof` command line uses the `-scaled` option by default, which can display profiles with CPU-cycle granularity and millisecond units if the procedures selected for display have short run times.

3 The `gprof` output comprises three sections: a call-graph profile, a timing profile, and an index (a concise means of identifying each procedure). In this example, the three sections have been separated by rows of asterisks (with the section names) that do not appear in the output produced by `gprof`. In the call-graph profile section, each routine in the program has its own subsection that is contained within dashed lines and identified by the index number in the first column.

4 This line describes the `main` routine, which is the subject of this portion of the call-graph profile because it is the leftmost routine in

the rightmost column of this section. The index number [2] in the first column corresponds to the index number [2] in the index section at the end of the output. The percentage in the second column reports the total amount of time in the subgraph that is accounted for by main and its descendants, in this case add_vector, mul_by_scalar, and print_it. The 1 in the called column indicates the total number of times that the main routine is called.

5 This line describes the relationship of add_vector to main. Because add_vector is below main in this section, add_vector is identified as the child of main. The fraction 100/100 indicates that of the total of 100 calls to add_vector (the denominator), 100 of these calls came from main (the numerator).

6 This line describes the relationship of main to add_vector. Because main is listed above add_vector in the last column, main is identified as the parent of add_vector.

For nonthreaded programs, hiprof can alternatively count the number of machine cycles used or page faults suffered by the program. The cost of each parent procedure's calls is accurately calculated, making the call-graph profile much more useful than the one in the default mode, which can only estimate the costs of parents. Also, the CPU time (in nanosecond units for short tests) or page-faults count reported for the instrumented routines includes that for the uninstrumented routines that they call. This can summarize the costs and reduce the run-time overhead, but note that the machine-cycle counter wraps if no instrumented procedure is called at least every few seconds.

In the following example, the hiprof command's -cycles option is used to display a profile of the machine cycles used by the sample program:

```
% cc -o libsample.so -shared -g1 -O2 add_vector.c mul_by_scalar.c print_it.c
% cc -o sample -g1 -O2 profsample.c -L. -lsample -lm
% hiprof -cycles -numbers -L. -inc libsample.so sample
```

The resulting sample profile is shown in Example 8–3.

Example 8–3: Sample hiprof -cycles Profile Using gprof

```
% cc -o libsample.so -shared -g1 -O2 add_vector.c mul_by_scalar.c print_it.c
add_vector.c:
mul_by_scalar.c:
print_it.c:

% cc -o sample -g1 -O2 profsample.c -L. -lsample -lm

% hiprof -cycles -numbers -L. -inc libsample.so sample
hiprof: info: instrumenting sample ...

hiprof: info: the LD_LIBRARY_PATH environment variable is not defined
hiprof: info: setting LD_LIBRARY_PATH=.:.
```

Example 8–3: Sample hiprof -cycles Profile Using gprof (cont.)

```
hiprof: info: running instrumented program sample.hiprof ...

Value = 179804.149985

hiprof: info: instrumented program exited with status 0
hiprof: info: displaying profile for sample ...
˝
gprof -b -scaled -all -numbers -L. sample ./sample.hiout

granularity: cycles; units: seconds*1e-9; total: 362320292e-9 seconds

                                    called /     total    parents
index  %total   self  descendents   called +      self  name     index
                                    called /     total    children

                                                          <spontaneous>
[1]    100.0    893310 361426982         1              __start [1]
                       361371860         1 /         1    main [2]

------------------------------------------------------

                       361371860         1 /         1    __start [1]
[2]     99.7  36316218 325055642         1              main [2]
                       321107275       100 /       100    add_vector [3]
                         3519530         2 /         2    mul_by_scalar [4]
                          428838         1 /         1    print_it [5]

------------------------------------------------------

                       321107275       100 /       100    main [2]
[3]     88.6 321107275         0       100              add_vector [3]

------------------------------------------------------

                         3519530         2 /         2    main [2]
[4]      1.0   3519530         0         2              mul_by_scalar [4]

------------------------------------------------------

                          428838         1 /         1    main [2]
[5]      0.1    428838         0         1              print_it [5]

------------------------------------------------------

granularity: cycles; units: seconds*1e-9; total: 362320292e-9 seconds

  %   cumulative     self           self/call total/call
total     units     units     calls   seconds   seconds  name
 88.6 321107275 321107275       100   3211e-6   3211e-6 add_vector [3]
 10.0 357423492  36316218         1  36316e-6 361372e-6 main [2]
  1.0 360943022   3519530         2   1760e-6   1760e-6 mul_by_scalar [4]
  0.2 361836332    893310         1 893310e-9 362320e-6 __start [1]
  0.1 362265170    428838         1 428838e-9 428838e-9 print_it [5]
```

Example 8–3: Sample hiprof -cycles Profile Using gprof (cont.)

```
Index by function name - 11 entries

   [1] __start           <sample>
   [3] add_vector        <libsample.so>:"add_vector.c":1
   [2] main              <sample>:"profsample.c":12
   [4] mul_by_scalar     <libsample.so>:"mul_by_scalar.c":1
   [5] print_it          <libsample.so>:"print_it.c":4
```

8.3.2.1.2 Using the cc Command's -pg Option

The `cc` command's `-pg` option uses the same sampling technique as `hiprof`, but the program needs to be instrumented by compiling with the `-pg` option. (The program also needs to be compiled with the debug and optimization options indicated in Section 8.2.) Only the executable is profiled (not shared libraries), and few system libraries are compiled to generate a call-graph profile; therefore, `hiprof` may be preferred. However, the `cc` command's `-pg` option and `gprof` are supported in a very similar way on different vendors' UNIX systems, so this may be an advantage. For example:

```
 % cc -pg -o sample -g1 -O2 *.c -lm[1] % ./sample[2] % gprof
-scaled -b -numbers sample[3]
```

[1] The `cc` command's `-pg` call-graph profiling option creates an instrumented version of the program, `sample`. (You must specify the `-pg` option for both the compile and link phases.)

[2] Running the instrumented program produces a profiling data file (named `gmon.out`, by default) to be used by the `gprof` tool. For information about working with multiple data files, see Section 8.7.

[3] The `gprof` command extracts information from the profiling data file and displays it. The `-scaled` option displays CPU time in units that give the best precision without exceeding the report's column widths. The `-b` option suppresses the printing of a description of each field in the profile.

The resulting sample profile is shown in Example 8–4. The `gprof` tool estimates the total cost of calling a procedure (including its calls) in its call-graph profile.

Example 8–4: Sample cc -pg Profile Using gprof

```
% cc -pg -o sample -g1 -O2 add_vector.c mul_by_scalar.c print_it.c profsample.c -lm
add_vector.c:
mul_by_scalar.c:
print_it.c:
profsample.c:

% ./sample
```

Example 8–4: Sample cc -pg Profile Using gprof (cont.)

```
Value = 179804.149985

% gprof -scaled -b -numbers sample

granularity: samples per 8 bytes; units: seconds*1e-3; total: 326e-3 seconds

                                    called /     total    parents
index  %total   self  descendents   called +      self  name      index
                                    called /     total    children

                    5       321         1 /         1    __start [2]
[1]    100.0        5       321         1               main [1]
                  317         0       100 /       100    add_vector [3]
                    4         0         2 /         2    mul_by_scalar [4]
                    0         0         1 /         1    print_it [5]

------------------------------------------------------

                  317         0       100 /       100    main [1]
[3]     97.3      317         0       100               add_vector [3]

------------------------------------------------------

                    4         0         2 /         2    main [1]
[4]      1.2        4         0         2               mul_by_scalar [4]

------------------------------------------------------

                    0         0         1 /         1    main [1]
[5]      0.0        0         0         1               print_it [5]

------------------------------------------------------

granularity: samples per 8 bytes; units: seconds*1e-3; total: 326e-3 seconds

  %   cumulative     self           self/call total/call
total     units     units     calls   seconds   seconds  name
 97.3       317       317       100   3174e-6   3174e-6 add_vector [3]
  1.5       322         5         1   4883e-6 326172e-6 main [1]
  1.2       326         4         2   1953e-6   1953e-6 mul_by_scalar [4]
  0.0       326         0         1         0         0 print_it [5]

Index by function name - 4 entries

   [3] add_vector         <sample>:"add_vector.c":1
   [1] main               <sample>:"profsample.c":12
   [4] mul_by_scalar      <sample>:"mul_by_scalar.c":1
   [5] print_it           <sample>:"print_it.c":4
```

8.3.2.2 CPU–Time/Event Profiles for Sourcelines/Instructions

A good performance-improvement strategy may start with a procedure-level profile of the whole program (possibly with a call graph, to provide an overall picture), but it will often progress to detailed profiling of individual source lines and instructions. The following tools provide this information:

- The `uprofile` profiler (Section 8.3.2.2.1)

- The hiprof profiler (Section 8.3.2.2.2)
- The cc command's -p option (Section 8.3.2.2.3)
- The pixie profiler (Section 8.3.2.2.4)

8.3.2.2.1 Using the uprofile Profiler

The uprofile profiler (see uprofile(1)) uses a sampling technique to generate a profile of the CPU time or events, such as cache misses, associated with each procedure or source line or instruction. The sampling frequency depends on the processor type and the statistic being sampled, but it is on the order of a millisecond for CPU time. The profiler achieves this without modifying the application program at all, by using hardware counters that are built into the Alpha CPU. Running the uprofile command with no arguments yields a list of all the kinds of events that a particular machine can profile, depending on the nature of its architecture. The default is to profile machine cycles, resulting in a CPU-time profile. The following example shows how to display a profile of the instructions that used the top 90 percent of CPU time:

```
% cc -o sample -g1 -O2 *.c -lm 1
% uprofile -asm -quit 90cum% sample 2
```

The resulting sample profile, which includes explanatory text, is shown in Example 8–5.

1 For information about the -g1 and -O2 options, see Section 8.2.

2 The uprofile command runs the sample program, collecting the performance counter data into a profile data file (named umon.out, by default). Because prof options (-asm and -quit) are specified, uprofile then automatically runs the prof tool to display the profile.

The -asm option provides per-instruction profiling of cycles (and other CPU statistics, such as data cache misses, if specified). Because no counter statistics are specified here, uprofile displays a CPU-time profile for each instruction. The -quit 90cum% option truncates the profile after 90 percent of the whole has been printed (see Section 8.6.3). (Also available are the -heavy option, which reports the lines that executed the most instructions, and the -lines option, which reports the profile per source line within each procedure (see Section 8.6.2).

Example 8–5: Sample uprofile CPU-Time Profile Using prof

```
% cc -o sample -g1 -O2 add_vector.c mul_by_scalar.c print_it.c profsample.c -lm
add_vector.c:
mul_by_scalar.c:
print_it.c:
profsample.c:

% uprofile -asm -quit 90cum% sample
```

Example 8–5: Sample uprofile CPU-Time Profile Using prof (cont.)

```
Value = 179804.149985
Writing umon.out

Displaying profile for sample:

Profile listing generated Thu Dec  3 10:29:25 1998 with:
   prof -asm -quit 90cum% sample umon.out

----------------------------------------------------------------------------
*  -a[sm] using performance counters:                                      *
*    cycles0: 1 sample every 65536 Cycles (0.000164 seconds)               *
*  sorted in descending order by total time spent in each procedure;       *
*  unexecuted procedures excluded                                          *
----------------------------------------------------------------------------

Each sample covers 4.00 byte(s) for 0.052% of 0.3123 seconds

      millisec      %  cum %      address:line  instruction

add_vector (add_vector.c)
           0.0   0.00   0.00  0x120001260:2     addl    zero, a2, a2
           0.0   0.00   0.00  0x120001264:5     bis     zero, zero, t0
           0.0   0.00   0.00  0x120001268:5     ble     a2, 0x12000131c
           0.0   0.00   0.00  0x12000126c:5     subl    a2, 0x3, t1
           0.0   0.00   0.00  0x120001270:5     cmple   t1, a2, t2
           0.0   0.00   0.00  0x120001274:5     beq     t2, 0x1200012f4
           0.0   0.00   0.00  0x120001278:5     ble     t1, 0x1200012f4
           0.0   0.00   0.00  0x12000127c:5     subq    a0, a1, t3
           0.0   0.00   0.00  0x120001280:5     lda     t3, 31(t3)
           0.0   0.00   0.00  0x120001284:5     cmpule  t3, 0x3e, t3
           0.0   0.00   0.00  0x120001288:5     bne     t3, 0x1200012f4
           0.0   0.00   0.00  0x12000128c:5     ldq_u   zero, 0(sp)
          20.2   6.45   6.45  0x120001290:6     ldl     zero, 128(a1)
           1.3   0.42   6.87  0x120001294:6     lds     $f31, 128(a0)
          35.6  11.39  18.26  0x120001298:6     ldt     $f0, 0(a1)
          20.0   6.40  24.66  0x12000129c:6     ldt     $f1, 0(a0)
           9.7   3.10  27.75  0x1200012a0:6     ldt     $f10, 8(a1)
          14.9   4.77  32.53  0x1200012a4:6     ldt     $f11, 8(a0)
          17.4   5.56  38.09  0x1200012a8:6     ldt     $f12, 16(a1)
           7.0   2.26  40.35  0x1200012ac:6     ldt     $f13, 16(a0)
           8.2   2.62  42.97  0x1200012b0:6     ldt     $f14, 24(a1)
          12.9   4.14  47.11  0x1200012b4:6     ldt     $f15, 24(a0)
          24.9   7.97  55.09  0x1200012b8:6     addt    $f1,$f0,$f0
          24.7   7.92  63.01  0x1200012bc:6     addt    $f11,$f10,$f10
          37.8  12.12  75.13  0x1200012c0:6     addt    $f13,$f12,$f12
          39.2  12.54  87.67  0x1200012c4:6     addt    $f15,$f14,$f14
           0.8   0.26  87.93  0x1200012c8:5     addl    t0, 0x4, t0
           0.0   0.00  87.93  0x1200012cc:5     lda     a1, 32(a1)
           8.4   2.68  90.61  0x1200012d0:6     stt     $f0, 0(a0)
```

By comparison, the following example shows how to display a profile of the instructions that suffered the top 90 percent of data cache misses on an EV56 Alpha system:

```
% cc -o sample -g1 -O2 *.c -lm
% uprofile -asm -quit 90cum% dcacheldmisses sample
```

The resulting sample profile is shown in Example 8–6.

Example 8–6: Sample uprofile Data-Cache-Misses Profile Using prof

```
% uprofile -asm -quit 90cum% dcacheldmisses sample
Value = 179804.149985
Writing umon.out

Displaying profile for sample:

Profile listing generated Thu Dec  3 10:34:25 1998 with:
   prof -asm -quit 90cum% sample umon.out

----------------------------------------------------------------------------
*  -a[sm] using performance counters:                                      *
*    dcacheldmiss: 1 sample every 16384 DCache LD Misses[1]               *
*  sorted in descending order by samples recorded for each procedure;      *
*  unexecuted procedures excluded                                          *
----------------------------------------------------------------------------

Each sample covers 4.00 byte(s) for 0.18% of 550 samples[2]

       samples      %  cum %      address:line  instruction

add_vector (add_vector.c)
           0.0   0.00   0.00  0x120001260:2     addl    zero, a2, a2
           0.0   0.00   0.00  0x120001264:5     bis     zero, zero, t0
           0.0   0.00   0.00  0x120001268:5     ble     a2, 0x12000131c
           0.0   0.00   0.00  0x12000126c:5     subl    a2, 0x3, t1
           0.0   0.00   0.00  0x120001270:5     cmple   t1, a2, t2
           0.0   0.00   0.00  0x120001274:5     beq     t2, 0x1200012f4
           0.0   0.00   0.00  0x120001278:5     ble     t1, 0x1200012f4
           0.0   0.00   0.00  0x12000127c:5     subq    a0, a1, t3
           0.0   0.00   0.00  0x120001280:5     lda     t3, 31(t3)
           0.0   0.00   0.00  0x120001284:5     cmpule  t3, 0x3e, t3
           0.0   0.00   0.00  0x120001288:5     bne     t3, 0x1200012f4
           0.0   0.00   0.00  0x12000128c:5     ldq_u   zero, 0(sp)
           1.0   0.18   0.18  0x120001290:6     ldl     zero, 128(a1)
           3.0   0.55   0.73  0x120001294:6     lds     $f31, 128(a0)
          62.0  11.27  12.00  0x120001298:6     ldt     $f0, 0(a1)[3]
          64.0  11.64  23.64  0x12000129c:6     ldt     $f1, 0(a0)
           8.0   1.45  25.09  0x1200012a0:6     ldt     $f10, 8(a1)
          47.0   8.55  33.64  0x1200012a4:6     ldt     $f11, 8(a0)
          13.0   2.36  36.00  0x1200012a8:6     ldt     $f12, 16(a1)
          38.0   6.91  42.91  0x1200012ac:6     ldt     $f13, 16(a0)
          15.0   2.73  45.64  0x1200012b0:6     ldt     $f14, 24(a1)
          51.0   9.27  54.91  0x1200012b4:6     ldt     $f15, 24(a0)
          49.0   8.91  63.82  0x1200012b8:6     addt    $f1,$f0,$f0
         142.0  25.82  89.64  0x1200012bc:6     addt    $f11,$f10,$f10
          13.0   2.36  92.00  0x1200012c0:6     addt    $f13,$f12,$f12
```

[1] The stated sampling rate of 1 sample every 16384 means that each sample shown in the profile occurred after 16384 data cache misses, but not all of these occurred at the instruction indicated.

[2] The total number of samples is shown, not the number of data cache misses.

[3] Indicates the number of samples recorded at an instruction, which statistically implies a proportionate number of data cache misses.

The uprofile profiling technique has the advantage of very low run-time overhead. Also, the detailed information it can provide on the costs of executing individual instructions or source lines is essential in identifying exactly which operation in a procedure is slowing the program down.

The disadvantages of uprofile are as follows:

- Only executables can be profiled. To profile code in a library, you must first link the program with the -non_shared option.
- Only one program can be profiled with the hardware counters at one time.
- Threads cannot be profiled individually.
- The Alpha EV6 architecture's execution of instructions out of sequence can significantly reduce the accuracy of fine-grained profiles.

8.3.2.2.2 Using the hiprof Profiler

As noted earlier, the hiprof command's default PC-sampling technique can also generate CPU-time profiles like those of uprofile, using a sampling frequency of a little under a millisecond. The profile is not as accurate, because the call counting affects the performance, but it has some advantages:

- Shared libraries can be profiled.
- Threads can be individually profiled (at the cost of very large memory and data file size).
- It is independent of hardware resources and architecture.

In the following example, the hiprof command's -lines option is used to display a profile of the CPU time used by each source line, grouped by procedure:

```
% cc -o libsample.so -shared -g1 -O2 add_vector.c mul_by_scalar.c print_it.c
% cc -o sample -g1 -O2 profsample.c -L. -lsample -lm
% hiprof -lines -numbers -L. -inc libsample.so sample
```

The resulting sample profile is shown in Example 8–7.

Example 8–7: Sample hiprof -lines PC-Sampling Profile

```
% cc -o libsample.so -shared -g1 -O2 add_vector.c mul_by_scalar.c print_it.c
add_vector.c:
mul_by_scalar.c:
print_it.c:

% cc -o sample -g1 -O2 profsample.c -L. -lsample -lm

% hiprof -lines -numbers -L. -inc libsample.so sample
hiprof: info: instrumenting sample ...
```

Example 8–7: Sample hiprof -lines PC-Sampling Profile (cont.)

```
hiprof: info: the LD_LIBRARY_PATH environment variable is not defined
hiprof: info: setting LD_LIBRARY_PATH=.:.
hiprof: info: running instrumented program sample.hiprof ...

Value = 179804.149985

hiprof: info: instrumented program exited with status 0
hiprof: info: displaying profile for sample ...

gprof -b -scaled -all -lines -numbers -L. sample.hiprof ./sample.hiout

Milliseconds per source line, in source order within functions

procedure (file)                           line bytes    millisec     %  cum %

add_vector (add_vector.c)                     2    52         0.0  0.00   0.00
                                              5    92         1.0  0.30   0.30
                                              6    92       318.4 98.19  98.49
                                              8     4         0.0  0.00  98.49
mul_by_scalar (mul_by_scalar.c)               2    52         0.0  0.00  98.49
                                              5    72         0.0  0.00  98.49
                                              6    64         3.9  1.20  99.70
                                              8     4         0.0  0.00  99.70
main (profsample.c)                           9    32         0.0  0.00  99.70
                                             12   128         0.0  0.00  99.70
                                             16     8         0.0  0.00  99.70
                                             19    32         0.0  0.00  99.70
                                             20    36         0.0  0.00  99.70
                                             21    16         0.0  0.00  99.70
                                             22    24         0.0  0.00  99.70
                                             24     4         0.0  0.00  99.70
                                             25    36         0.0  0.00  99.70
                                             26    16         0.0  0.00  99.70
                                             27    28         1.0  0.30 100.00
                                             28    20         0.0  0.00 100.00
                                             29    40         0.0  0.00 100.00
                                             30     4         0.0  0.00 100.00
                                             31    20         0.0  0.00 100.00
                                             32     4         0.0  0.00 100.00
                                             33    20         0.0  0.00 100.00
                                             34    56         0.0  0.00 100.00
```

8.3.2.2.3 Using the cc Command’s -p Option

The `cc` command’s `–p` option uses a low-frequency sampling technique to generate a profile of CPU time that is similar to `uprofile`’s but statistically less accurate. However, the `-p` option does offer the following advantages:

- Shared libraries can be profiled
- Threads can be individually profiled
- Independent of hardware resources and architecture
- Common to many UNIX operating systems
- On Tru64 UNIX, can profile all the shared libraries used by a program

The program needs to be relinked with the –p option, but it does not need to be recompiled from source so long as the original compilation used an acceptable debug level, such as the –g1 cc command option (see Section 8.2). For example, to profile individual source lines and procedures of a program (if the program runs for long enough to generate a dense sample array):

```
% cc -p -o sample -g1 -O2 *.c -lm 1 % setenv PROFFLAGS '-all -stride 1' 2 %
./sample 3 % prof -all -proc -heavy -numbers sample 4
```

1 The cc command's -p PC-sample-profiling option creates an instrumented version of the program, called sample.

2 The -all option specified with the PROFFLAGS environment variable asks for all shared libraries to be profiled (see Section 8.6.4). This causes sqrt (from libm.so) to show up in the profile as the second highest CPU-time user. The variable must be set before the instrumented program is run.

  The -stride 1 option in PROFFLAGS asks for a separate counter to be used for each instruction, to allow accurate per-source-line profiling with prof's -heavy option.

3 Running the instrumented program produces a PC-sampling data file called mon.out, by default, to be used by the prof tool. For information about working with multiple data files, see Section 8.7.

4 The prof tool (see prof(1)) uses the PC-sampling data file to produce the profile. Because this technique works by periodic sampling of the program counter, you might see different output when you profile the same program multiple times.

  When running prof manually, as in this example, you can filter which shared libraries to include in the displayed profile; the -all option tells prof to include all libraries (see Section 8.6.4). The -proc[edures] option profiles the instructions executed in each procedure and the calls to procedures. The -heavy option reports the lines that executed the most instructions. (Also, -lines shows per-line profiles and -asm shows per-instruction profiles, both grouped by procedure.)

The resulting sample profile is shown in Example 8–8.

Example 8–8: Sample cc -p Profile Using prof

```
% cc -p -o sample -g1 -O2 add_vector.c mul_by_scalar.c print_it.c profsample.c -lm
add_vector.c:
mul_by_scalar.c:
print_it.c:
profsample.c:

% setenv PROFFLAGS "-all -stride 1"

% ./sample
Value = 179804.149985
```

Example 8–8: Sample cc -p Profile Using prof (cont.)

```
% prof -all -proc -heavy -numbers sample
Profile listing generated Mon Feb 23 15:33:07 1998 with:
   prof -all -proc -heavy -numbers sample

----------------------------------------------------------------------------
*  -p[rocedures] using pc-sampling;                                        *
*  sorted in descending order by total time spent in each procedure;       *
*  unexecuted procedures excluded                                          *
----------------------------------------------------------------------------

Each sample covers 4.00 byte(s) for 0.25% of 0.3955 seconds

%time     seconds  cum %   cum sec  procedure (file)

 93.1      0.3682   93.1      0.37 add_vector (<sample>:"add_vector.c":1)
  5.4      0.0215   98.5      0.39 sqrt (<libm.so>)
  1.0      0.0039   99.5      0.39 mul_by_scalar (<sample>:"mul_by_scalar.c":1)
  0.5      0.0020  100.0      0.40 main (<sample>:"profsample.c":12)

″
----------------------------------------------------------------------------
*  -h[eavy] using pc-sampling;                                             *
*  sorted in descending order by time spent in each source line;           *
*  unexecuted lines excluded                                               *
----------------------------------------------------------------------------

Each sample covers 4.00 byte(s) for 0.25% of 0.3955 seconds

procedure (file)                           line bytes   millisec      %  cum %

add_vector (add_vector.c)                     6    80      363.3  91.85  91.85
add_vector (add_vector.c)                     5    96        4.9   1.23  93.09
mul_by_scalar (mul_by_scalar.c)               6    60        3.9   0.99  94.07
main (profsample.c)                          20    36        2.0   0.49  94.57
```

8.3.2.2.4 Using the pixie Profiler

The `pixie` tool (see `pixie`(1)) can also profile source-lines and instructions (including shared libraries), but note that when it displays counts of `cycles`, it is actually reporting counts of instructions executed, not machine cycles. Its –`truecycles 2` option can estimate the number of cycles that would be used if all memory accesses were satisfied by the cache, but programs can rarely cache enough of their data for this to be accurate, and only the Alpha EV4 and EV5 families can be fully simulated in this way. For example:

```
% cc -o sample -g1 -O2 *.c -lm 1
% pixie -all -proc -heavy -quit 5 sample 2
```

1 For information about the `-g1` and `-O2` options, see Section 8.2.

2 The `pixie` command creates an instrumented version of the program (`sample.pixie`) and an instruction-addresses file (`sample.Addrs`). Because `prof` options (`-proc, -heavy, -quit`) are specified, `pixie` automatically runs that instrumented program to create an

instructions-counts file (sample.Counts) and then runs prof to display the profile, using the .Addrs and .Counts files as input.

The -all option asks for shared libraries to be profiled. Although a pixie profile can include shared libraries, system libraries (like libm.so, which contains the sqrt function) do not include source-line numbers, so they are not included in the -heavy option's per-line profile, and static procedures get proc_at... names created for them.

The -heavy option reports lines that executed the most instructions. The -proc[edures] option profiles the instructions executed in each procedure and the calls to procedures. The -quit 5 option truncates the report after 5 lines for each mode (-heavy, -proc[edures]).

The resulting sample profile is shown in Example 8–9.

Example 8–9: Sample pixie Profile Using prof

```
% cc -o sample -g1 -O2 add_vector.c mul_by_scalar.c print_it.c profsample.c -lm
add_vector.c:
mul_by_scalar.c:
print_it.c:
profsample.c:

% pixie -all -proc -heavy -quit 5 sample
pixie: info: instrumenting sample ...

pixie: info: the LD_LIBRARY_PATH environment variable is not defined
pixie: info: setting LD_LIBRARY_PATH=.
pixie: info: running instrumented program sample.pixie ...

Value = 179804.149985

pixie: info: instrumented program exited with status 0
pixie: info: displaying profile for sample ...
˝
Profile listing generated Mon Feb 23 15:33:55 1998 with:
   prof -pixie -all -procedures -heavy -quit 5 sample ./sample.Counts

----------------------------------------------------------------------------
*  -p[rocedures] using basic-block counts;                                 *
*  sorted in descending order by the number of cycles executed in each     *
*  procedure; unexecuted procedures are excluded                           *
----------------------------------------------------------------------------

69089823 cycles (0.1727 seconds at 400.00 megahertz)

    cycles %cycles  cum %  seconds     cycles  bytes procedure (file)
                                        /call  /line

  60001400   86.85  86.85   0.1500     600014     48 add_vector (add_vector.c)
   7100008   10.28  97.12   0.0178         72      ? sqrt (<libm.so>)
   1301816    1.88  99.01   0.0033    1301816     27 main (profsample.c)
    675020    0.98  99.98   0.0017     337510     36 mul_by_scalar (mul_by_scalar.c)
       854    0.00  99.98   0.0000        854      ? __cvtas_t_to_a (<libc.so>)
˝
----------------------------------------------------------------------------
*  -p[rocedures] using invocation counts;                                  *
*  sorted in descending order by number of calls per procedure;            *
*  unexecuted procedures are excluded                                      *
```

Example 8–9: Sample pixie Profile Using prof (cont.)

```
----------------------------------------------------------------------------

100504 invocations total

     calls  %calls    cum%    bytes procedure (file)

    100000   99.50   99.50      820 sqrt (<libm.so>)
       100    0.10   99.60      192 add_vector (add_vector.c)
        39    0.04   99.64      164 proc_at_0x3ff815bc1e0 (<libc.so>)
        38    0.04   99.67      144 proc_at_0x3ff815bc150 (<libc.so>)
        38    0.04   99.71       16 proc_at_0x3ff815bc140 (<libc.so>)
˝
----------------------------------------------------------------------------
*  -h[eavy] using basic-block counts;                                      *
*  sorted in descending order by the number of cycles executed in each     *
*  line; unexecuted lines are excluded                                     *
----------------------------------------------------------------------------

procedure (file)                           line bytes     cycles      %  cum %

add_vector (add_vector.c)                     6    92   45000000  65.13  65.13
add_vector (add_vector.c)                     5    92   15001200  21.71  86.85
main (profsample.c)                          20    36     600003   0.87  87.71
main (profsample.c)                          22    24     600000   0.87  88.58
mul_by_scalar (mul_by_scalar.c)               6    64     487500   0.71  89.29
```

The `-procedures` profile includes a profile of call counts.

The optional `dxprof` command provides a graphical display of profiles collected by either `pixie` or the `cc` command's `–p` option.

You can also run the `prof -pixstats` command on the executable file `sample` to generate a detailed report on opcode frequencies, interlocks, a miniprofile, and more. For more information, see `prof`(1).

8.4 Minimizing System Resource Usage

The following sections describe the techniques and tools to help you minimize use of system resources by your application.

8.4.1 Techniques

The techniques described in the previous sections can improve an application's use of just the CPU. You can make further performance enhancements by improving the efficiency with which the application uses the other components of the computer system, such as heap memory, disk files, network connections, and so on.

As with CPU profiling, the first phase of a resource usage improvement process is to monitor how much memory, data I/O and disk space, elapsed time, and so on, is used. The throughput of the computer can then be increased or tuned in ways that help the program, or the program's design can be tuned to make better use of the computer resources that are available. For example:

- Reduce the size of the data files that the program reads and writes.
- Use memory-map files instead of regular I/O.
- Allocate memory incrementally on demand instead of allocating at startup the maximum that could be required.
- Fix heap leaks and do not leave allocated memory unused.

See the *System Configuration and Tuning* manual for a broader discussion of analyzing and tuning a Tru64 UNIX system.

8.4.2 Tools and Examples

The following sections discuss using system monitors and the Third Degree tool to minimize system resource usage.

8.4.2.1 System Monitors

The Tru64 UNIX base system commands `ps u`, `swapon -s`, and `vmstat 3` can show the currently active processes' usage of system resources such as CPU time, physical and virtual memory, swap space, page faults, and so on. See `ps`(1), `swapon`(8), and `vmstat`(3) for more information.

The optional `pview` command provides a graphical display of similar information for the processes that comprise an application. See `pview`(1).

The `time` commands provided by the Tru64 UNIX system and command shells provide an easy way to measure the total elapsed and CPU times for a program and it descendants. See `time`(1).

Performance Manager is an optional system performance monitoring and management tool with a graphical interface. See `pmgr`(8X).

For more information about related tools, see the *System Configuration and Tuning* manual.

8.4.2.2 Heap Memory Analyzers

The Third Degree tool (see `third`(1)) reports heap memory leaks in a program, by instrumenting it with the Third Degree memory-usage checker, running it, and displaying a log of leaks detected at program exit. For example:

```
% cc -o sample -g -non_shared *.c -lm 1
% third -display sample 2
```

1 Full debug information (that is, compiling with the `-g` option) is usually best for Third Degree, but if less is available, the reports will just be machine-level instead of source-level. The `-g1` option is fine if you are just checking for memory leaks.

2 The `third` command creates an instrumented version of the program (`sample.third`). Because the `-display` option is specified, `third` automatically runs that instrumented program to create a log file (`sample.3log`) and then runs `more` to display the file.

The resulting sample log file is shown in Example 8–10.

Example 8–10: Sample third Log File

```
cc -o sample -g -non_shared add_vector.c mul_by_scalar.c print_it.c profsample.c -lm
add_vector.c:
mul_by_scalar.c:
print_it.c:
profsample.c:
third -display sample
third: info: instrumenting sample ...
third: info: running instrumented program sample.third ...

Value = 179804.149985

third: info: instrumented program exited with status 0
third: info: error log for sample ...
more ./sample.3log
Third Degree version 5.0
sample.third run by jpx on frumpy.abc.dec.com at Mon Jun 21 16:59:50 1999

//////////////////////////////  Options  //////////////////////////////
-----------------------------------------------------------------------

-----------------------------------------------------------------------

------------------------------------------------------------------------
------------------------------------------------------------------------
New blocks in heap after program exit

Leaks - blocks not yet deallocated but apparently not in use:
 * A leak is not referenced by static memory, active stack frames,
   or unleaked blocks, though it may be referenced by other leaks.
 * A leak "not referenced by other leaks" may be the root of a leaked tree.
 * A block referenced only by registers, unseen thread stacks, mapped memory,
   or uninstrumented library data is falsely reported as a leak. Instrumenting
   shared libraries, if any, may reduce the number of such cases.
 * Any new leak lost its last reference since the previous heap report, if any.

A total of 800000 bytes in 1 leak were found:

800000 bytes in 1 leak (including 1 not referenced by other leaks) created at:
    malloc                         sample
    main                           sample, profsample.c, line 19
    __start                        sample

Objects - blocks not yet deallocated and apparently still in use:
```

Example 8–10: Sample third Log File (cont.)

```
 * An object is referenced by static memory, active stack, or other objects.
 * A leaked block may be falsely reported as an object if a pointer to it
   remains when a new stack frame or heap block reuses the pointer’s memory.
   Using the option to report uninitialized stack and heap may avoid such cases.
 * Any new object was allocated since the previous heap report, if any.

A total of 0 bytes in 0 objects were found:

------------------------------------------------------------------------
------------------------------------------------------------------------
                memory layout at program exit
                           heap     933888 bytes [0x140012000-0x1400f6000]
                          stack     819504 bytes [0x11ff37ed0-0x120000000]
                    sample data      66464 bytes [0x140000000-0x1400103a0]
                    sample text     802816 bytes [0x120000000-0x1200c4000]
=============================================================================
```

By default, Third Degree profiles memory leaks, with little overhead.

If you are interested only in leaks occurring during the normal operation of the program, not during startup or shutdown, you can specify additional places to check for previously unreported leaks. For example, the preshutdown leak report will give this information:

```
% third -display -after startup -before shutdown sample
```

Third Degree can also detect various kinds of bugs that may be affecting the correctness or performance of a program. See Chapter 7 for more information on debugging and leak detection.

The optional `dxheap` command provides a graphical display of Third Degree’s heap and bug reports. See `dxheap`(1).

The optional `mview` command provides a graphical analysis of heap usage over time. This view of a program’s heap can clearly show the presence (if not the cause) of significant leaks or other undesirable trends such as wasted memory. See `mview`(1).

8.5 Verifying the Significance of Test Cases

The following sections discuss the techniques and tools used to verify the significance of test cases.

8.5.1 Techniques

Most of the profiling techniques described in the previous sections are effective only if you profile and optimize or tune the parts of the program that are executed in the scenarios whose performance is important. Careful selection of the data used for the profiled test runs is often sufficient, but you may want a quantitative analysis of which code was and was not executed in a given set of tests.

8.5.2 Tools and Examples

The `pixie` command's –`t[estcoverage]` option reports lines of code that were not executed in a given test run. For example:

```
% cc -o sample -g1 -O2 *.c -lm
% pixie -t sample
```

Similarly, the –`zero` option reports the names of procedures that were never executed. Conversely, `pixie`'s –`p[rocedure]`, –`h[eavy]`, and –`a[sm]` options show which procedures, source lines, and instructions were executed.

If multiple test runs are needed to build up a typical scenario, the `prof` command can be run separately on a set of profile data files. For example:

```
% cc -o sample -g1 -O2 *.c -lm 1
% pixie -pids sample 2
% ./sample.pixie 3
% ./sample.pixie
% prof -pixie -t sample sample.Counts.* 4
```

1 For information about the `-g1` and `-O2` options, see Section 8.2.

2 The `pixie` command creates an instrumented version of the program (`sample.pixie`) and an instruction-addresses file (`sample.Addrs`). The `-pids` option adds the process ID of the instrumented program's test run (item 3) to the name of the profiling data file produced, so that a unique file is retained after each run. For information about working with multiple data files, see Section 8.7.

3 The instrumented program is run twice (usually with different input data) to produce two profiling data files named `sample.Counts.`*`pid`*.

4 The `-pixie` option tells `prof` to use `pixie` mode rather than the default PC-sampling mode. The `prof` tool uses the `sample.Addrs` file and the two `sample.Counts.`*`pid`* files to create the profile from the two runs.

8.6 Selecting Profiling Information to Display

Depending on the size of the application and the type of profiling you request, profilers may generate a very large amount of output. However, you are often only interested in profiling data about a particular portion of your application. The following sections show how you can select the appropriate information by using:

- `prof` options (with the `pixie`, `uprofile`, or `prof` command)
- `gprof` options (with the `hiprof` or `gprof` command)

Many of the `prof` and `gprof` options perform similar functions and have similar names and syntax. The examples used show `prof` options. For complete details, see hiprof(1), pixie(1), uprofile(1), prof(1), and gprof(1).

See Section 8.9 for information on using monitor routines to control profiling.

8.6.1 Limiting Profiling Display to Specific Procedures

The prof command's –only option prints profiling information for only a specified procedure. You can use this option several times on the command line. For example:

```
% pixie -only mul_by_scalar -only add_vector sample
```

The –exclude option prints profiling information for all procedures except the specified procedure. You can use this option several times on the command line. Do not use –only and –exclude together on the same command line.

Many of the prof profiling options print output as percentages; for example, the percentage of total execution time attributed to a particular procedure.

By default, the –only and –exclude options cause prof to calculate percentages based on all of the procedures in the application even if they were omitted from the listing. You can change this behavior with the –Only and –Exclude options. They work the same as –only and –exclude, but cause prof to calculate percentages based only on those procedures that appear in the listing.

The –totals option, used with the –procedures and –invocations listings, prints cumulative statistics for the entire object file instead of for each procedure in the object.

8.6.2 Displaying Profiling Information for Each Source Line

The prof and gprof -heavy and -lines options display the number of machine cycles, instructions, page faults, cache misses, and so on for each source line in your application. The -asm option displays them for each instruction.

The –heavy option prints an entry for every source line. Entries are sorted from the line with the heaviest use to the line with the lightest. Because –heavy often prints a large number of entries, you might want to use one of the –quit, –only, or –exclude options to reduce output to a manageable size (see Section 8.6.3).

The –lines option is similar to –heavy, but sorts the output differently. This option prints the lines for each procedure in the order that they occur in the source file. Even lines that never executed are printed. The procedures themselves are sorted by decreasing total use.

8.6.3 Limiting Profiling Display by Line

The –`quit` option reduces the amount of profiling output displayed. It affects the output from the –`procedures`, –`heavy`, and –`lines` profiling modes.

The –`quit` option has the following three forms:

–quit *N*	Truncates the listing after *N* lines.
–quit *N*%	Truncates the listing after the first line that shows less than *N*% of the total.
–quit *N*cum%	Truncates the listing after the line at which the cumulative total reaches *N*% of the total.

If you specify several modes on the same command line, –`quit` affects the output from each mode. The following command prints only the 20 most time-consuming procedures and the 20 most time-consuming source lines:

```
% pixie -procedures -heavy -quit 20 sample
```

Depending on the profiling mode, the total can refer to the total amount of time, the total number of executed instructions, or the total number of invocation counts.

8.6.4 Including Shared Libraries in the Profiling Information

When you are using the `hiprof`, `pixie`, `third`, `prof`, or `gprof` commands, the following options let you selectively display profiling information for shared libraries used by the program:

- –`all` displays profiles for all shared libraries (if any) described in the data file(s) in addition to the executable.
- –`incobj` *`lib`* displays the profile for the named shared library in addition to the executable.
- –`excobj` *`lib`* does not display the profile for the named shared library, if `-all` was specified.

For example, the following command displays profiling information in all shared libraries except for `user1.so`:

```
% prof -all -excobj user1.so sample
```

When you are using the `cc` command's `-p` option to obtain a PC-sampling profile, you can use the `PROFFLAGS` environment variable to include or exclude profiling information for shared libraries when the instrumented program is run (as shown in Example 8–8). The PROFFLAGS variable accepts the -all, -incobj, and excobj options.

For more information specific to shared libraries, see Section 8.6.4.1.

8.6.4.1 Specifying the Location of Instrumented Shared Libraries

The LD_LIBRARY_PATH environment variable is used to tell the loader where to find instrumented shared libraries.

By default, when you run hiprof, pixie, or third, the LD_LIBRARY_PATH variable is automatically set to point to the working directory. You can set it to point to an arbitrary directory, as in the following C shell example:

```
% setenv LD_LIBRARY_PATH "my_inst_libraries"
```

8.7 Merging Profile Data Files

If the program you are profiling is fairly complicated, you may want to run it several times with different inputs to get an accurate picture of its profile. Each of the profiling tools lets you merge profile data from different runs of a program. But first you must override the default behavior whereby each run of the instrumented program overwrites the existing data file. Section 8.7.1 explains how to do that. Section 8.7.2 explains how to merge data files.

8.7.1 Data File-Naming Conventions

The default name of a profiling data file depends on the tool used to create it, as follows:

Tool	Default Name of Profiling Data File
hiprof	*program*.hiout
pixie	*program*.Counts (used with *program*.Addrs)
uprofile	umon.out
cc -p	mon.out
cc -pg	gmon.out

By default, when you make multiple profiling runs, each run overwrites the existing data file in the working directory. Each tool has options for renaming data files so they can be preserved from run to run. The hiprof, pixie, and uprofile commands have the following options:

-dirname *path* Specifies a directory where the data file is to be created.

-pids Adds the process ID of the instrumented program's run to the data file name.

For example, the following command sequence produces two data files of the form `sample.`*`pid`*`.hiout` in the subdirectory `profdata`:

```
% hiprof -dir profdata -pids sample
% ./sample
% ./sample
```

Then, when using the `gprof` command, you can specify a wildcard to include all the profiling data files in the directory:

```
% gprof -b sample profdata/*
```

When using the `cc -p` or `cc -pg` profiling options, you need to set the `PROFFLAGS` environment variable (before running the instrumented program). For example, the following command sequence would produce two data files of the form *`pid`*`.sample` in the subdirectory `profdata` (C shell example):

```
% cc -pg -o sample -g1 -O2 *.c -lm
% setenv PROFFLAGS "-dir profdata -pids"
% ./sample
% ./sample
```

Note that `PROFFLAGS` affects the profiling behavior of a program during its execution; it has no effect on the `prof` and `gprof` postprocessors. When you set `PROFFLAGS` to a null string, no profiling occurs.

For more information about file-naming conventions, see the tool reference pages. See Section 8.8 for the file-naming conventions for multithreaded programs.

8.7.2 Data File-Merging Techniques

After creating several profiling data files from multiple runs of a program, you can display profiling information based on an average of these files.

Use the `prof` or `gprof` postprocessor, depending on the profiling technique used, as follows:

Profiling Tool	Post Processor to Use
`cc -p`, `uprofile`, or `pixie`	`prof`
`cc -pg`, or `hiprof`	`gprof`

One merge technique is to specify the name of each data file explicitly on the command line. For example, the following command prints profiling information from two profile data files generated using `hiprof`:

```
% gprof sample sample.1510.hiout sample.1522.hiout
```

Keeping track of many different profiling data files, however, can be difficult. Therefore, prof and gprof provide the −merge option to combine several data files into a single merged file. When prof operates in pixie mode (prof −pixie), the −merge option combines the .Counts files. When prof operates in PC-sampling mode, this switch combines the mon.out or other profile data files.

The following example combines two profile data files into a single data file named total.out:

```
% prof -merge total.out sample 1773.sample 1777.sample
```

At a later time, you can then display profiling data using the combined file, just as you would use a normal mon.out file. For example:

```
%  prof -procedures sample total.out
```

The merge process is similar for −pixie mode. You must specify the executable file's name, the .Addrs file, and each .Counts file. For example:

```
% prof -pixie -merge total.Counts a.out a.out.Addrs \
  a.out.Counts.1866 a.out.Counts.1868
```

8.8 Profiling Multithreaded Applications

Note

To analyze a multithreaded appplication for potential logic and performance problems, you can use Visual Threads, which is available on the Tru64 UNIX Associated Products Volume 1 installation media. Visual Threads can be used on POSIX Threads Library applications and on Java applications. See dxthreads(1).

Profiling multithreaded applications is essentially the same as profiling nonthreaded applications. However, to profile multithreaded applications, you must compile your program with the −pthread or −threads option (as they are defined by the cc command). Other threads packages are not supported.

When using hiprof(1), pixie(1), or third(1), specify the -pthread option to enable the tool's thread-safe profiling support. The uprofile(1) command and the cc command's -p and -pg options need no extra option to become thread-safe.

The default case for profiling multithreaded applications is to provide one profile covering all threads. In this case, you get profiling across the entire process, and you get one output file of profiling data from all threads.

When using `hiprof`(1) or `pixie`(1), specify the `-threads` option for per-thread data.

When using the `cc` command's `-p` or `-pg` option, set the `PROFFLAGS` environment variable to `–threads` for per-thread profiling, as shown in the following C shell example:

```
setenv PROFFLAGS "-threads"
```

The `uprofile`(1) and `third`(1) tools do not provide per-thread data.

Per-thread profiling data files are given names that include unique thread numbers, which reflect the sequence in which the threads were created or in which their profiling was started.

If you use the `monitor( )` or `monstartup( )` calls (see Section 8.9) in a threaded program, you must first set `PROFFLAGS` to `"-disable_default -threads"`, which gives you complete control of profiling the application.

If the application uses `monitor( )` and allocates separate buffers for each thread profiled, you must first set `PROFFLAGS` to `"-disable_default -threads"` because this setting affects the file-naming conventions that are used. Without the `–threads` option, the buffer and address range used as a result of the first `monitor` or `monstartup` call would be applied to every thread that subsequently requests profiling. In this case, a single data file that covers all threads being profiled would be created.

Each thread in a process must call the `monitor( )` or `monstartup( )` routines to initiate profiling for itself.

8.9 Using monitor Routines to Control Profiling

The default behavior for the `cc` command's `-p` and `-pg` modes on Tru64 UNIX systems is to profile the entire text segment of your program and place the profiling data in `mon.out` for `prof` profiling or in `gmon.out` for `gprof` profiling. For large programs, you might not need to profile the entire text segment. The `monitor` routines provide the ability to profile portions of your program specified by the lower and upper address boundaries of a function address range.

The `monitor` routines are as follows:

`monitor`
Use this routine to gain control of explicit profiling by turning profiling on and off for a specific text range. This routine is not supported for `gprof` profiling.

`monstartup`
Similar to `monitor` except it specifies address range only and is supported for `gprof` profiling.

moncontrol — Use this routine with monitor and monstartup to turn PC sampling on or off during program execution for a specific process or thread.

monitor_signal — Use this routine to profile nonterminating programs, such as daemons.

You can use monitor and monstartup to profile an address range in each shared library as well as in the static executable.

For more information on these functions, see monitor(3).

By default, profiling begins as soon your program starts to execute. To prevent this behavior, set the PROFFLAGS environment variable to –disable_default, as shown in the following C shell example:

```
setenv PROFFLAGS "-disable_default"
```

Then, you can use the monitor routines to begin profiling after the first call to monitor or monstartup.

Example 8–11 demonstrates how to use the monstartup and monitor routines within a program to begin and end profiling.

Example 8–11: Using monstartup() and monitor()

```
/*  Profile the domath( ) routine using monstartup.
 *  This example allocates a buffer for the entire program.
 *  Compile command: cc -p foo.c -o foo -lm
 *  Before running the executable, enter the following
 *  from the command line to disable default profiling support:
 *  setenv PROFFLAGS -disable_default
 */

 #include <stdio.h>
 #include <sys/syslimits.h>

 char dir[PATH_MAX];

 extern void __start( );
 extern unsigned long _etext;

 main( )
 {
     int i;
     int a = 1;

     /* Start profiling between __start (beginning of text)
      * and _etext (end of text).  The profiling library
      * routines will allocate the buffer.
```

Example 8–11: Using monstartup() and monitor() (cont.)

```
         */

        monstartup(__start,&_etext);

           for(i=0;i<10;i++)
                domath( );

        /* Stop profiling and write the profiling output file. */

        monitor(0);
   }
    domath( )
   {
      int i;
      double d1, d2;

      d2 = 3.1415;
      for (i=0;  i<1000000;  i++)
         d1 = sqrt(d2)*sqrt(d2);
   }
```

The external name `_etext` lies just above all the program text. See `end`(3) for more information.

When you set the `PROFFLAGS` environment variable to `–disable_default`, you disable default profiling buffer support. You can allocate buffers within your program, as shown in Example 8–12.

Example 8–12: Allocating Profiling Buffers Within a Program

```
   /*  Profile the domath routine using monitor().
    *  Compile command: cc -p foo.c -o foo -lm
    *  Before running the executable, enter the following
    *  from the command line to disable default profiling support:
    *  setenv PROFFLAGS -disable_default
    */

   #include <sys/types.h>
   #include <sys/syslimits.h>

   extern char *calloc( );

   void domath(void);
   void nextproc(void);

   #define INST_SIZE 4          /* Instruction size on Alpha */
```

Example 8–12: Allocating Profiling Buffers Within a Program (cont.)

```
 char dir[PATH_MAX];

 main( )
 {
      int i;
      char *buffer;
      size_t bufsize;

      /*  Allocate one counter for each instruction to
       *  be sampled. Each counter is an unsigned short.
       */

      bufsize = (((char *)nextproc - (char *)domath)/INST_SIZE)
       * sizeof(unsigned short);

       /*  Use calloc( ) to ensure that the buffer is clean
        *  before sampling begins.
        */

       buffer = calloc(bufsize,1);

       /*  Start sampling.  */
        monitor(domath,nextproc,buffer,bufsize,0);
        for(i=0;i<10;i++)
              domath( );

       /* Stop sampling and write out profiling buffer.  */
                monitor(0);
}
 void domath(void)
{
  int i;
  double d1, d2;

     d2 = 3.1415;
     for (i=0;  i<1000000;  i++)
         d1 = sqrt(d2)*sqrt(d2);
 }

 void nextproc(void)
 {}
```

You use the `monitor_signal( )` routine to profile programs that do not terminate. Declare this routine as a signal handler in your program and build the program for `prof` or `gprof` profiling. While the program is executing, send a signal from the shell by using the `kill` command.

When the signal is received, monitor_signal is invoked and writes profiling data to the data file. If the program receives another signal, the data file is overwritten.

Example 8–13 shows how to use the monitor_signal routine.

Setting the PROFFLAGS environment variable to -sigdump SIGNAME provides the same capability without needing any new program code.

Example 8–13: Using monitor_signal() to Profile Nonterminating Programs

```
/* From the shell, start up the program in background.
 * Send a signal to the process, for example: kill -30 <pid>
 * Process the [g]mon.out file normally using gprof or prof
 */

#include <signal.h>

extern int monitor_signal();

main()
{
    int i;
    double d1, d2;

    /*
     * Declare monitor_signal() as signal handler for SIGUSR1
     */
    signal(SIGUSR1,monitor_signal);
    d2 = 3.1415;
    /*
     * Loop infinitely (absurd example of non-terminating process)
     */
    for (;;)
        d1 = sqrt(d2)*sqrt(d2);
}
```

9

Using and Developing Atom Tools

Program analysis tools are extremely important for computer architects and software engineers. Computer architects use them to test and measure new architectural designs, and software engineers use them to identify critical pieces of code in programs or to examine how well a branch prediction or instruction scheduling algorithm is performing. Program analysis tools are needed for problems ranging from basic block counting to data cache simulation. Although the tools that accomplish these tasks may appear quite different, each can be implemented simply and efficiently through code instrumentation.

Atom provides a flexible code instrumentation interface that is capable of building a wide variety of tools. Atom separates the common part in all problems from the problem-specific part by providing machinery for instrumentation and object-code manipulation, and allowing the tool designer to specify what points in a program are to be instrumented. Atom is independent of any compiler and language because it operates on object modules that make up a complete program.

This chapter discusses the following topics:

- How to run installed Atom tools and new Atom tools that are still under development (Section 9.1)
- How to develop specialized Atom tools (Section 9.2)

9.1 Running Atom Tools

The following sections describe how to:

- Use installed Atom tools (Section 9.1.1)
- Test Atom tools under development (Section 9.1.2)

9.1.1 Using Installed Tools

The Tru64 UNIX operating system provides a number of example Atom tools, listed in Table 9–1, to help you develop your own custom-designed Atom tools. These tools are distributed in source form to illustrate Atom's programming interfaces — they are not intended for production use. Section 9.2 describes some of the tools in more detail.

Table 9–1: Example Prepackaged Atom Tools

Tool	Description
`branch`	Instruments all conditional branches to determine how many are predicted correctly.
`cache`	Determines the cache miss rate if an application runs in an 8-KB direct-mapped cache.
`dtb`	Determines the number of dtb (data translation buffer) misses if the application uses 8-KB pages and a fully associative translation buffer.
`dyninst`	Provides fundamental dynamic counts of instructions, loads, stores, blocks, and procedures.
`inline`	Identifies potential candidates for inlining.
`iprof`	Prints the number of times each procedure is called as well as the number of instructions executed by each procedure.
`malloc`	Records each call to the `malloc` function and prints a summary of the application's allocated memory.
`prof`	Prints the number of instructions executed by each procedure in `pthread` programs.
`ptrace`	Prints the name of each procedure as it is called.
`replace`	Calls a replaced entry point in the application from an analysis routine.
`trace`	Generates an address trace, logs the effective address of every load and store operation, and logs the address of the start of every basic block as it is executed.

The example tools can be found in the `/usr/lib/cmplrs/atom/examples` directory. Each one has three files:

- An instrumentation file — a C source file that uses Atom's API to modify application programs such that additional routines provided by the tool are invoked at particular times during program execution.
- An analysis file — a C source file that contains the routines that are invoked by the modified program when it is executed. These analysis routines can collect the run-time data that the tool reports.
- A description file (`*toolname*.desc`) — a text file that tells Atom the names of the tool's instrumentation and analysis files, along with any options that Atom should use when running the tool.

Atom tools that are put into production use or that are delivered to customers as products usually have `.o` object modules installed instead of their proprietary sources. The Tru64 UNIX `hiprof`(1), `pixie`(1), and `third`(1) commands and the Visual Threads product include Atom tools that are delivered, installed, and run this way. By convention, their instrumentation, analysis, and description files are in `/usr/lib/cmplrs/atom/tools`.

To run an installed Atom tool or example on an application program, use the following form of the `atom`(1) command:

atom *application_program* –tool *toolname* [–env *environment*] [*options...*]

This form of the `atom` command requires the `–tool` option and accepts the `–env` option.

The `–tool` option identifies the installed Atom tool to be used. By default, Atom searches for installed tools in the `/usr/lib/cmplrs/atom/tools` and `/usr/lib/cmplrs/atom/examples` directories. You can add directories to the search path by supplying a colon-separated list of additional directories to the `ATOMTOOLPATH` environment variable.

The `–env` option indicates that an alternative version of the tool is desired. For example, some Tru64 UNIX tools require `-env threads` to run the thread-safe version. The `atom`(1) command searches for a *`toolname.env.`*`desc` file instead of the default *`toolname`*`.desc` file. It prints an error message if a description file for the specified environment cannot be found.

9.1.2 Testing Tools Under Development

A second form of the `atom`(1) command is provided to make it easy to compile and run a new atom tool that you are developing. You just name the instrumentation and analysis files directly on the command line:

atom application_program *instrumentation_file* [*analysis_file*] [*options...*]

This form of the command requires the *`instrumentation_file`* parameter and accepts the *`analysis_file`* parameter, but not the `-tool` or `-env` options.

The *`instrumentation_file`* parameter specifies the name of a C source file or an object module that contains the Atom tool's instrumentation procedures. If the instrumentation procedures are in more than one file, the `.o` of each file may be linked together into one file using the `ld` command with a `-r` option. By convention, most instrumentation files have the suffix `.inst.c` or `.inst.o`.

If you pass an object module for this parameter, consider compiling the module with either the `-g1` or`-g` option. If there are errors in your instrumentation procedures, Atom can issue more complete diagnostic messages when the instrumentation procedures are thus compiled.

The *analysis_file* parameter specifies the name of a C source file or an object module that contains the Atom tool's analysis procedures. If the analysis routines are in more than one file, the `.o` of each file may be linked together into one file using the `ld` command with a `-r` option. Note that you do not need to specify an analysis file if the instrumentation file does not call analysis procedures to the application it instruments. By convention, most analysis files have the suffix `.anal.c` or `.anal.o`.

Analysis routines may perform better if they are compiled as a single compilation unit.

You can have multiple instrumentation and analysis source files. The following example creates composite instrumentation and analysis objects from several source files:

```
% cc -c file1.c file2.c
% cc -c file7.c file8
% ld -r -o tool.inst.o file1.o file2.o
% ld -r -o tool.anal.o file7.o file8.o
% atom hello tool.inst.o tool.anal.o -o hello.atom
```

Note

You can also write analysis procedures in C++. You must assign a type of `extern "C"` to each procedure to allow it to be called from the application. You must also compile and link the analysis files before entering the `atom` command. For example:

```
% cxx -c tool.a.C
% ld -r -o tool.anal.o tool.a.o -lcxx -lexc
% atom hello tool.inst.c tool.anal.o -o hello.atom
```

9.1.3 Atom Options

With the exception of the –`tool` and –`env` options, both forms of the `atom` command accept any of the remaining options described in `atom`(1). The following options deserve special mention:

`-A1`

Causes Atom to optimize calls to analysis routines by reducing the number of registers that need to be saved and restored. For some tools, specifying this option increases the performance of the instrumented application by a factor of two (at the expense of some increase in application size). The default behavior is for Atom not to apply these optimizations.

`-all`

Instruments all statically loaded shared libraries in the shared executable.

`-debug`

Lets you debug instrumentation routines with the `dbx` debugger. Atom transfers control to the symbolic debugger at the start of the instrumentation routine. In the following example, the `ptrace` sample tool is run under the `dbx` debugger. The instrumentation is stopped at line 12, and the procedure name is printed.

```
% atom hello ptrace.inst.c ptrace.anal.c -o hello.ptrace -debug
dbx version 3.11.8
Type 'help' for help.
Stopped in InstrumentAll
(dbx) stop at 12
[4] stop at "/udir/test/scribe/atom.user/tools/ptrace.inst.c":12
(dbx) c
[3] [InstrumentAll:12 ,0x12004dea8] if (name == NULL) name = "UNKNOWN";
(dbx) p name
0x2a391 = "__start"
```

`-ladebug`

Lets you debug instrumentation routines with the `ladebug` debugger. Atom transfers control to `ladebug` at the start of the instrumentation routine. Use `ladebug` if the instrumentation routines are threaded or contain C++ code. See the *Ladebug Debugger Manual* for more information.

`-excobj`

Excludes the named shared library from instrumentation. You can use the `-excobj` option more than once to specify several shared libraries.

`-fork`

Specifies that fork support is required. Use this option to avoid deadlocks in multithreaded applications.

`-ga` (`-g`)

Produces the instrumented program with debugging information. This option lets you debug analysis routines with a symbolic debugger. The default `-A0` option (not `-A1`) is recommended with `-ga` (or `-g`). For example:

```
% atom hello ptrace.inst.c ptrace.anal.c -o hello.ptrace -ga
% dbx hello.ptrace
dbx version 3.11.8
Type 'help' for help.
(dbx) stop in ProcTrace
[2] stop in ProcTrace
```

```
(dbx) r
[2] stopped at [ProcTrace:5 ,0x120005574] fprintf (stderr,"%s\n",name);
(dbx) n
__start
    [ProcTrace:6 ,0x120005598] }
```

-gap

Produces the instrumented program with debugging information. This enables debugging of analysis and application routines. The prefix "_APP_" is attached to all variable and procedure names in the application. The default -A0 option (not -A1) is recommended when -gpa is used.

-gp

Produces the instrumented program with debugging information. This option lets you debug application routines with a symbolic debugger.

-gpa

Produces the instrumented program with debugging information. This enables debugging of analysis and application routines. The prefix "_ANA_" is attached to all variable and procedure names in the analysis object. The default -A0 option (not -A1) is recommended when -gpa is used.

-heapbase

Changes the base of the analysis heap. Use the -heapbase option if the default analysis heap location conflicts with an address range used by the application program. For the new base, you can choose either a new hex address location, a default 31-bit address location, or the first page after the end of the application's bss segment.

-ii

Allows reuse (or incremental instrumentation) of a previously instrumented shared library.

-incobj

Instruments the named shared library. You can use the -incobj option more than once to specify several shared libraries.

-keep

Indicates that the temporary files that Atom creates are to be placed in the current working directory and not deleted when instrumentation is complete.

`-L`

Changes the library directory search order for shared object libraries so that Atom never looks for them in the default library directories. Use this option when the default library directories should not be searched and only the directories specified by `-Ldir` are to be searched.

`-L`*dir*

Changes the library directory search order for shared object libraries so that Atom searches for them in *`dir`* before searching the default library directories. You can specify multiple `-L`*dir* options to specify several directory names.

`-map`

Produces a list of the starting addresses of the sections in the instrumented executable.

`-o`

Specifies a name for the executable output file.

`–pthread`

Specifies that thread-safe support is required. Use this option when instrumenting threaded applications.

`-shlib`*dir*

Specifies an existing directory to which Atom writes the instrumented shared libraries.

`-suffix`

Specifies a filename suffix that is appended to the name of each object when Atom writes the instrumented version.

`-toolargs`

Passes arguments to the Atom tool's instrumentation routine. Atom passes the arguments in the same way that they are passed to C programs, using the *`argc`* and *`argv`* arguments to the `main` program. For example:

```
#include <stdio.h>
unsigned InstrumentAll(int argc, char **argv) {
     int i;
     for (i = 0; i < argc; i++) {
       printf(stderr,"argv[%d]: %s\n",argv[i]);
     }
}
```

The following example shows how Atom passes the –toolargs arguments:

```
% atom hello args.inst.c -toolargs="8192 4"
argv[0]: hello
argv[1]: 8192
argv[2]: 4
```

`-v`

Displays each step Atom takes to create the instrumented program.

`-version`

Displays Atom's version number.

`-w0`

Displays all warning messages, including those that are normally suppressed.

`-w1`

Suppresses warning messages that can be safely ignored. This is the default.

`-w2`

Suppresses warning messages emitted when processing the analysis routines.

`-w3`

Suppresses warning messages about shared library processing errors.

`-Wla` and `-Wca`

Passes the specified options to the analysis file's link and compilation phases, respectively.

`-Wli` and `-Wci`

Passes the specified options to the instrumentation file's link and compilation phases, respectively.

9.2 Developing Atom Tools

The remainder of this chapter describes how to develop atom tools.

9.2.1 Atom's View of an Application

Atom views an application as a hierarchy of components:

1. The program, including the executable and all shared libraries.
2. A collection of objects. An object can be either the main executable or any shared library. An object has its own set of attributes (such as its name) and consists of a collection of procedures.
3. A collection of procedures, each of which consists of a collection of entry points and a collection of basic blocks.
4. A collection of basic blocks, each of which consists of a collection of instructions.
5. A collection of instructions.

Atom tools insert instrumentation points in an application program at procedure, entry point, basic block, or instruction boundaries. For example, basic block counting tools instrument the beginning of each basic block, data cache simulators instrument each load and store instruction, and branch prediction analyzers instrument each conditional branch instruction.

At any instrumentation point, Atom allows a tool to insert a call to an analysis routine. The tool can specify that the call be made before or after an object, procedure, entry point, basic block, or instruction.

9.2.2 Atom Instrumentation Routine

A tool's instrumentation routine contains the code that traverses the application's objects, procedures, entry points, basic blocks, and instructions to locate instrumentation points; adds calls to analysis procedures; and builds the instrumented version of an application.

As described in `atom_instrumentation_routines`(5), an instrumentation routine can employ one of the following interfaces based on the needs of the tool:

`Instrument (int` ***`iargc,`*** **`char **`*****`iargv,`*** **`Obj *`*****`obj`*****`)`**

Atom calls the `Instrument` routine for each object in the application program. As a result, an `Instrument` routine does not need to use the object navigation routines (such as `GetFirstObj`). Because Atom automatically writes each modified object before passing the next to the `Instrument` routine, the `Instrument` routine should never call the `BuildObj`, `WriteObj`, or `ReleaseObj` routine. When using the `Instrument` interface, you can define an `InstrumentInit` routine to perform tasks required before Atom calls `Instrument` for the first object (such as defining analysis routine prototypes, adding program level instrumentation calls, and performing global initializations). You can also define an InstrumentFini routine to perform tasks required after Atom calls Instrument for the last object (such as global cleanup).

InstrumentAll (int *iargc,* **char *****iargv***)**

Atom calls the InstrumentAll routine once for the entire application program, which allows a tool's instrumentation code itself to determine how to traverse the application's objects. With this method, there are no InstrumentInit or InstrumentFini routines. An InstrumentAll routine must call the Atom object navigation routines and use the BuildObj, WriteObj, or ReleaseObj routine to manage the application's objects.

Regardless of the instrumentation routine interface, Atom passes the arguments specified in the -toolargs option to the routine. In the case of the Instrument interface, Atom also passes a pointer to the current object.

9.2.3 Atom Instrumentation Interfaces

Atom provides a comprehensive interface for instrumenting applications. The interface supports the following types of activities:

- Navigating among a program's objects, procedures, entry points, basic blocks, and instructions. See Section 9.2.3.1.
- Building, releasing, and writing objects. See Section 9.2.3.2.
- Obtaining information about the different components of an application. See Section 9.2.3.3.
- Resolving names and call targets. See Section 9.2.3.4.
- Adding calls to analysis routines at desired locations in the program. See Section 9.2.3.5.
- Intercepting calls to entry points in a program. See Section 9.2.3.6.

9.2.3.1 Navigating Within a Program

The Atom application navigation routines, described in atom_application_navigation(5), allow an Atom tool's instrumentation routine to find locations in an application at which to add calls to analysis procedures as follows:

- The GetFirstObj, GetLastObj, GetNextObj, and GetPrevObj routines navigate among the objects of a program. For nonshared programs, there is only one object. For call-shared programs, the first object corresponds to the main program. The remaining objects are each of its dynamically linked shared libraries.

- The GetFirstObjProc and GetLastObjProc routines return a pointer to the first or last procedure, respectively, in the specified object. The GetNextProc and GetPrevProc routines navigate among the procedures of an object.
- The GetFirstEntry and GetLastEntry routines return a pointer to the first or last entry point, respectively, in the specified procedure. The GetNextEntry and GetPrevEntry routines navigate among the entry points of a procedure.
- The GetFirstBlock, GetLastBlock, GetNextBlock, and GetPrevBlock routines navigate among the basic blocks of a procedure.
- The GetFirstInst, GetLastInst, GetNextInst, and GetPrevInst routines navigate among the instructions of a basic block.
- The GetInstBranchTarget routine returns a pointer to the instruction that is the target of a specified branch instruction.
- The GetProcObj routine returns a pointer to the object that contains the specified procedure.
- The GetEntryProc and GetBlockProc routines return a pointer to the procedure that contains the specified entry point or basic block, respectively.
- The GetEntryBlock routine returns a pointer to the first basic block in the specified entry point. The GetInstBlock routine returns a pointer to the basic block that contains the specified instruction.

9.2.3.2 Building Objects

The Atom object management routines, described in atom_object_management(5), allow an Atom tool's InstrumentAll routine to build, write, and release objects.

The BuildObj routine builds the internal data structures Atom requires to manipulate the object. An InstrumentAll routine must call the BuildObj routine before traversing the procedures in the object and adding analysis routine calls to the object. The WriteObj routine writes the instrumented version of the specified object, deallocating the internal data structures the BuildObj routine previously created. The ReleaseObj routine deallocates the internal data structures for the given object, but it does not write out the instrumented version the object.

The IsObjBuilt routine returns a nonzero value if the specified object has been built with the BuildObj routine but not yet written with the WriteObj routine or unbuilt with the ReleaseObj routine.

9.2.3.3 Obtaining Information About an Application's Components

The Atom application query routines, described in `atom_application_query`(5), allow an instrumentation routine to obtain static information about a program and its objects, procedures, entry points, basic blocks, and instructions.

Table 9–2 lists the routines that provide information about a program.

Table 9–2: Atom Program Query Routines

Routine	Description
`GetAnalName`	Returns the name of the analysis file, as passed to the `atom` command. This routine is useful for tools that have a single instrumentation file and multiple analysis files.
`GetErrantShlibName`	Returns the name of the shared library that Atom was unable to process.
`GetErrantShlibErr`	Returns an error code to explain why Atom was unable to process the shared library.
`GetProgInfo`	Returns the number of objects in a program, the number of objects that the tool writer requested to be instrumented, or the number of shared libraries that Atom was unable to process.

Table 9–3 lists the routines that provide information about a program's objects.

Table 9–3: Atom Object Query Routines

Routine	Description
`GetObjInfo`	Returns information about an object's text, data, and bss segments; the number of procedures, entry points, basic blocks, or instructions it contains; its object ID; information about how the object was linked; or a Boolean hint as to whether the given object should be excluded from instrumentation.
`GetObjInstArray`	Returns an array consisting of the 32-bit instructions included in the object.
`GetObjInstCount`	Returns the number of instructions in the array included in the array returned by the `GetObjInstArray` routine.
`GetObjName`	Returns the original file name of the specified object.
`GetObjOutName`	Returns the name of the instrumented object.

The following instrumentation routine, which prints statistics about the program's objects, demonstrates the use of Atom object query routines:

```
#include <stdio.h>
#include <cmplrs/atom.inst.h>
unsigned InstrumentAll(int argc, char **argv)
{
   Obj *o; Proc *p;
   const unsigned int *textSection;
   long textStart;
   for (o = GetFirstObj(); o != NULL; o = GetNextObj(o)) {
     BuildObj(o);
     textSection = GetObjInstArray(o);
     textStart = GetObjInfo(o,ObjTextStartAddress);
     printf("Object %d\n", GetObjInfo(o,ObjID));
     printf("   Object name: %s\n", GetObjName(o));
     printf("   Text segment start: 0x%lx\n", textStart);
     printf("   Text size: %ld\n", GetObjInfo(o,ObjTextSize));
     printf("   Second instruction: 0x%x\n", textSection[1]);
     ReleaseObj(o);
   }
   return(0);
}
```

Because the instrumentation routine adds no procedures to the executable, there is no need for an analysis procedure. The following example demonstrates the process of compiling and instrumenting a program with this tool. A sample run of the instrumented program prints the object identifier, the compile-time starting address of the text segment, the size of the text segment, and the binary for the second instruction. The disassembler provides a convenient method for finding the corresponding instructions.

```
% cc hello.c -o hello
% atom hello info.inst.c -o hello.info
Object 0
  Object Name: hello
  Start Address: 0x120000000
  Text Size: 8192
  Second instruction: 0x239f001d
Object 1
  Object Name: /usr/shlib/libc.so
  Start Address: 0x3ff80080000
  Text Size: 901120
  Second instruction: 0x239f09cb
% dis hello | head -3
  0x120000fe0: a77d8010     ldq t12, -32752(gp)
  0x120000fe4: 239f001d     lda at, 29(zero)
  0x120000fe8: 279c0000     ldah at, 0(at)
% dis /ust/shlib/libc.so | head -3
  0x3ff800bd9b0: a77d8010   ldq t12,-32752(gp)
  0x3ff800bd9b4: 239f09cb   lda at,2507(zero)
  0x3ff800bd9b8: 279c0000   ldah at, 0(at)
```

Table 9–4 lists the routines that provide information about an object's procedures.

Table 9–4: Atom Procedure Query Routines

Routine	Description
`GetProcInfo`	Returns information pertaining to the procedure's stack frame, register-saving, register-usage, and prologue characteristics as defined in the *Calling Standard for Alpha Systems* manual and the *Assembly Language Programmer's Guide*. Such values are important to tools, like Third Degree, that monitor the stack for access to uninitialized variables. It can also return information about the procedure such as the number of entry points, basic blocks, or instructions it contains; its procedure ID and type of symbol resolution; its lowest or highest source line number; procedure offset to the first instruction; procedure offset to the instruction that modifies the stack pointer; whether it has any alternate entry points; whether it has any interprocedural branches or jumps; and whether its address has been taken.
`ProcGP`	Returns the global pointer (GP) of the procedure.
`ProcFileName`	Returns the name of the source file that contains the procedure.
`ProcName`	Returns the procedure's name.
`ProcPC`	Returns the compile-time program counter (PC) of the first instruction in the procedure.

Table 9–5 list the routines that provide information about an object's main and alternate entry points.

Table 9–5: Atom Entry Point Query Routines

Routine	Description
`EntryName`	Returns the entry point's name.
`EntryPC`	Returns the compile-time program counter (PC) of the first instruction of the entry point.
`GetEntryInfo`	Returns information about the entry point such as whether it is redundant.
`GetEntryProc`	Returns the enclosing procedure of the entry point.

Table 9–6 lists the routines that provide information about a procedure's basic blocks.

Table 9–6: Atom Basic Block Query Routines

Routine	Description
`BlockPC`	Returns the compile-time program counter (PC) of the first instruction in the basic block.
`GetBlockInfo`	Returns the number of instructions in the basic block or the block ID.
`IsBranchTarget`	Indicates if the block is the target of a branch instruction.

Table 9–7 lists the routines that provide information about a basic block's instructions.

Table 9–7: Atom Instruction Query Routines

Routine	Description
`GetInstBinary`	Returns a 32-bit binary representation of the assembly language instruction.
`GetInstClass`	Returns the instruction class (for example, floating-point load or integer store) as defined by the *Alpha Architecture Reference Manual*.
`GetInstInfo`	Parses the entire 32-bit instruction and obtains all or a portion of that instruction.
`GetInstRegEnum`	Returns the register type (floating-point or integer) from an instruction field as returned by the `GetInstInfo` routine.
`GetInstRegUsage`	Returns a bit mask with one bit set for each possible source register and one bit set for each possible destination register.
`InstLineNo`	Returns the instruction's source line number.
`InstPC`	Returns the compile-time program counter (PC) of the instruction.
`IsInstType`	Indicates whether the instruction is of the specified type (load instruction, store instruction, conditional branch, or unconditional branch).

9.2.3.4 Resolving Names and Call Targets

Atom's symbol resolution routines, described in atom_application_symbols(5), allow an Atom tool's instrumentation routine to find and get application objects, procedures, entry points, and instructions that are either named or targets of a call site:

- The `FindObj` and `FindObjDepthFirst` routines search (breadth-first and depth-first, respectively) for the named entry point in all of the objects and returns the object that contains the named entry point.

- The FindProc routine returns the procedure that contains a main or alternate entry point with the specified name in the given object.
- The FindEntry routine returns the main or alternate entry point with the specified name in the given object.
- The FindInst routine returns the first instruction at the address of the resolved symbol with the specified name in the given object
- The GetTargetName routine returns the name of the entry point that is the target of the procedure call.
- The GetTargetObj, GetTargetProc, and GetTargetEntry, routines return, respectively, the object, procedure, or entry point, that contains the target of the procedure call.
- The GetTargetInst routine returns the instruction to which the specified instruction jumps or branches.

9.2.3.5 Adding Calls to Analysis Routines

The Atom application instrumentation routines, described in atom_application_instrumentation(5), add arbitrary procedure calls at various points in the application as follows:

- You must use the AddCallProto routine to specify the prototype of each analysis procedure to be added to the program. In other words, an AddCallProto call must define the procedural interface for each analysis procedure used in calls to AddCallProgram, AddCallObj, AddCallProc, AddCallEntry, AddCallBlock, and AddCallInst. Atom provides facilities for passing application and analysis data into the added procedure as either constants, register contents, address translation structures, or computed values such as effective addresses and branch conditions.
- Use the AddCallProgram routine in an instrumentation routine to add a call to an analysis procedure before a program starts execution or after it completes execution. Typically such an analysis procedure does something that applies to the whole program, such as opening an output file or parsing command-line options.
- Use the AddCallObj routine in an instrumentation routine to add a call to an analysis procedure before an object starts execution or after it completes execution. Typically such an analysis procedure does something that applies to the single object, such as initializing some data for its procedures.
- Use the AddCallProc routine in an instrumentation routine to add a call to an analysis procedure before a procedure starts execution or after it completes execution.

- Use the AddCallEntry routine in an instrumentation routine to add a call to an analysis procedure before a main or alternate entry point starts execution.
- Use the AddCallBlock routine in an instrumentation routine to add a call to an analysis procedure before a basic block starts execution or after it completes execution.
- Use the AddCallInst routine in an instrumentation routine to add a call to an analysis procedure before a given instruction executes or after it executes.
- Use the ReplaceProcedure routine to replace a procedure in the instrumented program. For example, the Third Degree Atom tool replaces memory allocation functions such as malloc and free with its own versions to allow it to check for invalid memory accesses and memory leaks.

9.2.3.6 Intercepting Calls to Entry Points

Atom's call interception routines, described in atom_application_instrumentation(5), intercept an application's calls to a main or alternate entry point with a call to a replacement analysis routine as follows:

- You must use the ReplaceProto routine to specify the prototype of each replacement analysis routine to be called instead of the replaced entry point. In other words, a ReplaceProto call must define the procedural interface for each analysis routine referenced in calls to the ReplaceEntry routine. Atom provides facilities for passing application and analysis data into the replacement analysis routine as either constants, register contents, address translation structures, or computed values such as the run-time address of the replaced entry.
- Use the ReplaceEntry routine to replace a main or alternate entry point in the instrumented program with a call to a replacement analysis routine. A ReplaceEntry call must specify the entry point to replace, the replacement analysis routine, and the procedural interface arguments prototyped in ReplaceProto. Only the specified entry point is replaced. Other entry points in the same application can be replaced by using additional ReplaceEntry calls. The replacement routine performs the analysis and can emulate the replaced entry point by calling it using a computed ReplAddrValue value. For an example of this emulation, see Section 9.2.6.

For example, the following pair of ReplaceProto and ReplaceEntry calls intercepts memcpy(3) library calls by instead calling my_memcpy with three application arguments and two analysis arguments (the return address of the application's call to memcpy( ) and the replaced entry point's run-time address):

```
ReplaceProto ("my_memcpy(VALUE, VALUE, VALUE, REGV, VALUE)");
ReplaceEntry (FindEntry(obj,"memcpy"),  /* entry point to replace */
              "my_memcpy",              /* replacement analysis routine */
              ArgValue,                 /* application argument */
              ArgValue,                 /* application argument */
              ArgValue,                 /* application argument */
              REG_RA,                   /* analysis argument */
              ReplAddrValue));          /* analysis argument */
```

The associated replacement analysis routine would be declared with three application and two analysis arguments:

```
void * my_memcpy (void * s1, const void * s2, size_t n, long call_address,
                  void * (*memcpy_ptr) (void *,const void *,size_t));
```

9.2.4 Atom Description File

An Atom tool's description file, as described in `atom_description_file`(5), identifies and describes the tool's instrumentation and analysis files. It can also specify the options to be used by the `cc`, `ld`, and `atom` commands when it is compiled, linked, and invoked. Each Atom tool must supply at least one description file.

There are two types of Atom description file:

- A description file providing an environment for generalized use of the tool. A tool can provide only one general-purpose environment. The name of this type of description file has the following format:

  tool.desc

- A description file providing an environment for use of the tool in specific contexts, such as in a multithreaded application or in kernel mode. A tool can provide several special-purpose environments, each of which has its own description file. The name of this type of description file has the following format:

  tool.environment.desc

The names supplied for the *`tool`* and *`environment`* portions of these description file names correspond to values the user specifies to the `−tool` and `−env` options of an `atom` command when invoking the tool.

An Atom description file is a text file containing a series of tags and values. See `atom_description_file`(5) for a complete description of the file's syntax.

9.2.5 Writing Analysis Procedures

An instrumented application calls analysis procedures to perform the specific functions defined by an Atom tool. An analysis procedure can use system calls or library functions, even if the same call or function is instrumented within the application. The routines used by the analysis routine and the instrumented application are physically distinct. The following library routines can and cannot be called by analysis routines:

- Standard C Library (libc.a) routines (including system calls) can be called, except for:
  - unwind(3) routines and other exception-handling routines
  - tis(3) routines

  Also, the standard I/O routines have certain differences in behavior, as described in Section 9.2.5.1.
- The pthread_atfork(3) routine can be called only if the -fork option was used during program instrumentation.
- Math Library (libm.a) routines can be called.
- Other routines related to multithreading or exception-handling should not be called (for example, pthread(3), exc_*, and libmach routines).
- Other routines that assume a particular environment (for example, X and Motif) may not be useful or correct in an Atom analysis environment.

Thread Local Storage (TLS) is not supported in analysis routines.

9.2.5.1 Input/Output

The standard I/O library provided to analysis routines does not automatically flush and close streams when the instrumented program terminates, so the analysis code must flush or close them explicitly when all output has been completed. Also, the stdout and stderr streams that are provided to analysis routines will be closed when the application calls exit(), so analysis code may need to duplicate one or both of these streams if they need to be used after application exit (for example, in a ProgramAfter or ObjAfter analysis routine. See the prof tool described in Section 9.1.1 for an example of how to open an additional stream for I/O.

For output to stderr (or a duplicate of stderr) to appear immediately, analysis code should call setbuf(stream,NULL) to make the stream unbuffered or call fflush after each set of fprintf calls. Similarly, analysis routines using C++ streams can call cerr.flush().

9.2.5.2 fork and exec System Calls

If a process calls a fork function but does not call an exec function, the process is cloned and the child inherits an exact copy of the parent's state. In many cases, an Atom tool expects this behavior. For example, an instruction-address tracing tool sees references for both the parent and the child, interleaved in the order in which the references occurred.

In the case of an instruction-profiling tool (for example, the trace tool referenced in Table 9–1), the file is opened at a ProgramBefore instrumentation point and, as a result, the output file descriptor is shared between the parent and the child processes. If the results are printed at a ProgramAfter instrumentation point, the output file contains the parent's data, followed by the child's data (assuming that the parent process finishes first).

For tools that count events (for example, the prof tool referenced in Table 9–1), the data structures that hold the counts should be returned to zero in the child process after the fork call because the events occurred in the parent, not the child. This type of Atom tool can support correct handling of fork calls by instrumenting the fork library procedure and calling an analysis procedure with the return value of the fork routine as an argument. If the analysis procedure is passed a return value of 0 (zero) in the argument, it knows that it was called from a child process. It can then reset the counts variable or other data structures so that they tally statistics only for the child process.

9.2.6 Calling Application Entry Points That Have Been Replaced

Entry points that have been replaced may be called from the replacement analysis routine by using the ReplAddrValue parameter that was passed in during the ReplaceEntry call. The ReplAddrValue argument contains the address of the replaced entry point, as described in Section 9.2.3.6. This address provides a way for the replacement analysis routine to emulate the replaced entry point by simply calling it.

In the following example, the function memcpy( ) is replaced with the analysis function my_memcpy( ), which in turn calls the replaced memcpy( ) function.

The following source listing contains the instrumentation code:

```
#include <string.h>
#include <cmplrs/atom.inst.h>

unsigned InstrumentAll (int argc, char **argv)
{
    Xlate *     px;
    Obj *       o;
    Entry *     e;

    /*
     *  Prototype the replacement routine.
     */
    ReplaceProto("my_memcpy(VALUE, VALUE, VALUE, REGV, VALUE)");

    /*
     *  Resolve the object that contains memcpy().
     */
    o = FindObj("memcpy");
    if (o) {
        /*
         * Build the object containing memcpy so the memcpy entry point
         * can be resolved and replaced.
         */
        if (BuildObj(o)) return(1);

        /*
         * Resolve the memcpy entry point.
         */
        e = FindEntry(o, "memcpy");

        /*
         * Prefix the memcpy entry point with atom-generated code to
         * call the analysis routine my_memcpy instead.
         */
        ReplaceEntry(e, "my_memcpy",
                        ArgValue, ArgValue, ArgValue, REG_RA, ReplAddrValue);

        /*
         * Write the instrumented object.
         */
        WriteObj(o);
    }
    return (0);
}
```

The following source listing contains the analysis code:

```
#include <stdio.h>
#include <stdlib.h>
#include <cmplrs/atom.anal.h>

/*
 * Replacement routine for memcpy();
 */
void * my_memcpy (void * s1, const void * s2, size_t n, long call_address,
                  void * (*memcpy_ptr) (void *, const void *, size_t))
{
   void * ptr = 0;

   /*
    * Report the call.
    */
   printf ("memcpy called from %lx\n", call_address);

   /*
    * Call the original memcpy().
    */
   ptr = (*memcpy_ptr) (s1, s2, n);

   return (ptr);
}
```

9.2.7 Determining the Instrumented PC from an Analysis Routine

Atom’s address translation routines, described in Xlate(5), allow you to determine the instrumented program counter (PC) for selected instructions. You can use these functions to build a table that translates an instruction’s PC in the uninstrumented application to its PC in the instrumented application.

To provide the instruction's address in an address translation buffer, the instrumentation routine must first call the CreateXlate routine. After the address translation buffer has been created, the instrumentation routine adds an instruction's address to it by calling the AddXlateAddress routine. Addresses of entry points can also be added to an address translation buffer by calling the AddXlateEntry routine. An address translation buffer can only hold addresses from a single object.

An Atom tool's instrumentation routine passes an address translation buffer to an analysis routine by passing it as a parameter of type XLATE *, as indicated in the analysis routine's prototype definition in an AddCallProto call.

Another way to determine an instrumented PC is to specify a formal parameter type of REGV in an analysis routine's prototype and pass the REG_IPC value.

An Atom tool's analysis routine uses the following analysis interfaces to access an address translation buffer passed to it:

- The XlateNum routine returns the number of addresses in the specified address translation buffer.
- The XlateInstTextStart routine returns the starting address of the text segment for the instrumented object corresponding to the specified address translation buffer.
- The XlateInstTextSize routine returns the size of the text segment.
- The XlateLoadShift routine returns the difference between the run-time addresses in the object corresponding to the specified address translation buffer and the compile-time addresses.
- The XlateAddr routine returns the instrumented run-time address for the instruction in the specified position of the specified address translation buffer. Note that the run-time address for an instruction in a shared library is not necessarily the same as its compile-time address.

The following example demonstrates the use of the Xlate routines by the instrumentation and analysis files of a tool that uses the Xlate routines. This tool prints the target address of every jump instruction. To use it, enter the following command:

```
% atom progname xlate.inst.c xlate.anal.c -all
```

The following source listing (xlate.inst.c) contains the instrumentation for the xlate tool:

```
#include <stdlib.h>
#include <stdio.h>
#include <alpha/inst.h>
#include <cmplrs/atom.inst.h>
```

```
static void             address_add(unsigned long);
static unsigned         address_num(void);
static unsigned long *  address_paddrs(void);
static void             address_free(void);

void InstrumentInit(int iargc, char **iargv)
{
    /* Create analysis prototypes. */
    AddCallProto("RegisterNumObjs(int)");
    AddCallProto("RegisterXlate(int, XLATE *, long[0])");
    AddCallProto("JmpLog(long, REGV)");

    /* Pass the number of objects to the analysis routines. */
    AddCallProgram(ProgramBefore, "RegisterNumObjs",
        GetProgInfo(ProgNumberObjects));
}

Instrument(int iargc, char **iargv, Obj *obj)
{
    Proc *                      p;
    Block *                     b;
    Inst *                      i;
    Xlate *                     pxlt;
    union alpha_instruction     bin;
    ProcRes                     pres;
    unsigned long               pc;
    char                        proto[128];

    /*
     * Create an XLATE structure for this Obj.  We use this to translate
     * instrumented jump target addresses to pure jump target addresses.
     */
    pxlt = CreateXlate(obj, XLATE_NOSIZE);

    for (p = GetFirstObjProc(obj);  p;  p = GetNextProc(p)) {
        for (b = GetFirstBlock(p);  b;  b = GetNextBlock(b)) {
            /*
             * If the first instruction in this basic block has had its
             * address taken, it’s a potential jump target.  Add the
             * instruction to the XLATE and keep track of the pure address
             * too.
             */
            i = GetFirstInst(b);
            if (GetInstInfo(i, InstAddrTaken)) {
                AddXlateAddress(pxlt, i);
                address_add(InstPC(i));
            }

            for (;  i;  i = GetNextInst(i)) {
                bin.word = GetInstInfo(i, InstBinary);
                if (bin.common.opcode == op_jsr &&
                    bin.j_format.function == jsr_jmp)
                {
                    /*
                     * This is a jump instruction.  Instrument it.
                     */
                    AddCallInst(i, InstBefore, "JmpLog",  InstPC(i),
                        GetInstInfo(i, InstRB));
                }
            }
        }
    }

    /*
```

```
     * Re-prototype the RegisterXlate() analysis routine now that we
     * know the size of the pure address array.
     */
    sprintf(proto, "RegisterXlate(int, XLATE *, long[%d])", address_num());
    AddCallProto(proto);

    /*
     * Pass the XLATE and the pure address array to this object.
     */
    AddCallObj(obj, ObjBefore, "RegisterXlate", GetObjInfo(obj, ObjID),
        pxlt, address_paddrs());

    /*
     * Deallocate the pure address array.
     */
    address_free();
}

/*
** Maintains a dynamic array of pure addresses.
*/
static unsigned long *  pAddrs;
static unsigned         maxAddrs = 0;
static unsigned         nAddrs = 0;

/*
** Add an address to the array.
*/
static void address_add(
    unsigned long       addr)
{
    /*
     * If there’s not enough room, expand the array.
     */
    if (nAddrs >= maxAddrs) {
        maxAddrs = (nAddrs + 100) * 2;
        pAddrs = realloc(pAddrs, maxAddrs * sizeof(*pAddrs));
        if (!pAddrs) {
            fprintf(stderr, "Out of memory\n");
            exit(1);
        }
    }

    /*
     * Add the address to the array.
     */
    pAddrs[nAddrs++] = addr;
}


/*
** Return the number of elments in the address array.
*/
static unsigned address_num(void)
{
    return(nAddrs);
}


/*
** Return the array of addresses.
*/
static unsigned long *address_paddrs(void)
{
    return(pAddrs);
}

/*
** Deallocate the address array.
*/
static void address_free(void)
{
    free(pAddrs);
    pAddrs = 0;
    maxAddrs = 0;
    nAddrs = 0;
}
```

The following source listing (`xlate.anal.c`) contains the analysis routine for the `xlate` tool:

```
#include <stdlib.h>
#include <stdio.h>
#include <cmplrs/atom.anal.h>

/*
 * Each object in the application gets one of the following data
 * structures.  The XLATE contains the instrumented addresses for
 * all possible jump targets in the object.  The array contains
 * the matching pure addresses.
 */
typedef struct {
    XLATE *             pXlt;
    unsigned long *     pAddrsPure;
} ObjXlt_t;

/*
 * An array with one ObjXlt_t structure for each object in the
 * application.
 */
static ObjXlt_t *       pAllXlts;
static unsigned         nObj;
static int      translate_addr(unsigned long, unsigned long *);
static int      translate_addr_obj(ObjXlt_t *, unsigned long,
                    unsigned long *);

/*
**  Called at ProgramBefore.  Registers the number of objects in
**  this application.
*/
void RegisterNumObjs(
    unsigned    nobj)
{
    /*
     * Allocate an array with one element for each object.  The
     * elements are initialized as each object is loaded.
     */
    nObj = nobj;
    pAllXlts = calloc(nobj, sizeof(pAllXlts));
    if (!pAllXlts) {
        fprintf(stderr, "Out of Memory\n");
        exit(1);
    }
}

/*
**  Called at ObjBefore for each object.  Registers an XLATE with
**  instrumented addresses for all possible jump targets.  Also
**  passes an array of pure addresses for all possible jump targets.
*/
void RegisterXlate(
    unsigned            iobj,
    XLATE *             pxlt,
    unsigned long *     paddrs_pure)
{
    /*
     * Initialize this object’s element in the pAllXlts array.
     */
    pAllXlts[iobj].pXlt = pxlt;
    pAllXlts[iobj].pAddrsPure = paddrs_pure;
}

/*
**  Called at InstBefore for each jump instruction.  Prints the pure
**  target address of the jump.
*/
void JmpLog(
    unsigned long       pc,
    REGV                targ)
{
    unsigned long       addr;

    printf("0x%lx jumps to - ", pc);
    if (translate_addr(targ, &addr))
        printf("0x%lx\n", addr);
    else
        printf("unknown\n");
}

/*
**  Attempt to translate the given instrumented address to its pure
**  equivalent.  Set ’*paddr_pure’ to the pure address and return 1
**  on success.  Return 0 on failure.
**
**  Will always succeed for jump target addresses.
*/
static int translate_addr(
    unsigned long       addr_inst,
    unsigned long *     paddr_pure)
{
    unsigned long       start;
    unsigned long       size;
    unsigned            i;

    /*
     * Find out which object contains this instrumented address.
     */
    for (i = 0;  i < nObj;  i++) {
        start = XlateInstTextStart(pAllXlts[i].pXlt);
        size = XlateInstTextSize(pAllXlts[i].pXlt);
        if (addr_inst >= size && addr_inst < start + size) {
            /*
             * Found the object, translate the address using that
             * object’s data.
             */
            return(translate_addr_obj(&pAllXlts[i], addr_inst,
                paddr_pure));
        }
    }

    /*
     * No object contains this address.
     */
    return(0);
}

/*
**  Attempt to translate the given instrumented address to its
**  pure equivalent using the given object's translation data.
**  Set '*paddr_pure' to the pure address and return 1 on success.
**  Return 0 on failure.
*/
static int translate_addr_obj(
    ObjXlt_t *          pObjXlt,
    unsigned long       addr_inst,
    unsigned long *     paddr_pure)
{
    unsigned    num;
    unsigned    i;

    /*
     * See if the instrumented address matches any element in the XLATE.
     */
    num = XlateNum(pObjXlt->pXlt);
    for (i = 0;  i < num;  i++) {
        if (XlateAddr(pObjXlt->pXlt, i) == addr_inst) {
            /*
             * Matches this XLATE element, return the matching pure
             * address.
             */
            *paddr_pure = pObjXlt->pAddrsPure[i];
            return(1);
        }
    }

    /*
     * No match found, must not be a possible jump target.
     */
    return(0);
}
```

9.2.8 Sample Tools

This section describes the basic tool-building interface by using three simple examples: procedure tracing, instruction profiling, and data cache simulation.

9.2.8.1 Procedure Tracing

The ptrace tool prints the names of procedures in the order in which they are executed. The implementation adds a call to each procedure in the application. By convention, the instrumentation for the ptrace tool is placed in the file ptrace.inst.c. For example:

```
#include <stdio.h>
#include <cmplrs/atom.inst.h> 1

unsigned InstrumentAll(int argc, char **argv) 2
{
   Obj *o; Proc *p;
   AddCallProto("ProcTrace(char *)"); 3
   for (o = GetFirstObj(); o != NULL; o = GetNextObj(o)) { 4
```

```
 9      if (BuildObj(o)) return 1; 5
10      for (p = GetFirstObjProc(o); p != NULL; p = GetNextProc(p)) { 6
11        const char *name = ProcName(p); 7
12        if (name == NULL) name = "UNKNOWN"; 8
13        AddCallProc(p,ProcBefore,"ProcTrace",name); 9
14      }
15      WriteObj(o); 10
16    }
17    return(0);
18 }
```

1 Includes the definitions for Atom instrumentation routines and data structures.

2 Defines the InstrumentAll procedure. This instrumentation routine defines the interface to each analysis procedure and inserts calls to those procedures at the correct locations in the applications it instruments.

3 Calls the AddCallProto routine to define the ProcTrace analysis procedure. ProcTrace takes a single argument of type char *.

4 Calls the GetFirstObj and GetNextObj routines to cycle through each object in the application. If the program was linked nonshared, there is only a single object. If the program was linked call-shared, it contains multiple objects: one for the main executable and one for each dynamically linked shared library. The main program is always the first object.

5 Builds the first object. Objects must be built before they can be used. In very rare circumstances, the object cannot be built. The InstrumentAll routine reports this condition to Atom by returning a nonzero value.

6 Calls the GetFirstObjProc and GetNextProc routines to step through each procedure in the application program.

7 For each procedure, calls the ProcName procedure to find the procedure name. Depending on the amount of symbol table information that is available in the application, some procedures names, such as those defined as static, may not be available. (Compiling applications with the −g1 option provides this level of symbol information.) In these cases, Atom returns NULL.

8 Converts the NULL procedure name string to UNKNOWN.

9 Calls the AddCallProc routine to add a call to the procedure pointed to by p. The ProcBefore argument indicates that the analysis procedure is to be added before all other instructions in the procedure. The name of the analysis procedure to be called at this instrumentation point is ProcTrace. The final argument is to be passed to the analysis procedure. In this case, it is the procedure named obtained on line 11.

10 Writes the instrumented object file to disk.

The instrumentation file added calls to the ProcTrace analysis procedure. This procedure is defined in the analysis file ptrace.anal.c as shown in the following example:

```
#include <stdio.h>

void ProcTrace(char *name)
{
  fprintf(stderr, "%s\n",name);
}
```

The ProcTrace analysis procedure prints, to stderr, the character string passed to it as an argument. Note that an analysis procedure cannot return a value.

After the instrumentation and analysis files are specified, the tool is complete. To demonstrate the application of this tool, compile and link the following application as follows:

```
#include <stdio.h>
main()
{
   printf("Hello world!\n");
}
```

The following example builds a nonshared executable, applies the ptrace tool, and runs the instrumented executable. This simple program calls almost 30 procedures.

```
% cc -non_shared hello.c -o hello
% atom hello ptrace.inst.c ptrace.anal.c -o hello.ptrace
% hello.ptrace
     __start
     main
     printf
     _doprnt
     __getmbcurmax
     strchr
     strlen
     memcpy
     .
     .
     .
```

The following example repeats this process with the application linked call-shared. The major difference is that the LD_LIBRARY_PATH environment variable must be set to the current directory because Atom creates an instrumented version of the libc.so shared library in the local directory.

```
% cc hello.c -o hello
% atom hello ptrace.inst.c ptrace.anal.c -o hello.ptrace -all
% setenv LD_LIBRARY_PATH 'pwd'
% hello.ptrace
     __start
     _call_add_gp_range
     __exc_add_gp_range
     malloc
     cartesian_alloc
     cartesian_growheap2
     __getpagesize
     __sbrk
     .
     .
     .
```

The call-shared version of the application calls almost twice the number of procedures that the nonshared version calls.

Note that only calls in the original application program are instrumented. Because the call to the `ProcTrace` analysis procedure did not occur in the original application, it does not appear in a trace of the instrumented application procedures. Likewise, the standard library calls that print the names of each procedure are also not included. If the application and the analysis program both call the `printf` function, Atom would link into the instrumented application two copies of the function. Only the copy in the application program would be instrumented. Atom also correctly instruments procedures that have multiple entry points.

9.2.8.2 Profile Tool

The `iprof` example tool counts the number of instructions a program executes. It is useful for finding critical sections of code. Each time the application is executed, `iprof` creates a file called `iprof.out` that contains a profile of the number of instructions that are executed in each procedure and the number of times each procedure is called.

The most efficient place to compute instruction counts is inside each basic block. Each time a basic block is executed, a fixed number of instructions are executed. The following example shows how the `iprof` tool's instrumentation procedure (`iprof.inst.c`) performs these tasks:

```
#include <stdio.h>
#include <cmplrs/atom.inst.h>

static int n = 0;

static const char *     SafeProcName(Proc *);

void InstrumentInit(int argc, char **argv)
{
    AddCallProto("OpenFile(int)"); 1
    AddCallProto("ProcedureCalls(int)");
    AddCallProto("ProcedureCount(int,int)");
12    AddCallProto("ProcedurePrint(int,char*)");
13    AddCallProto("CloseFile()");
14    AddCallProgram(ProgramAfter,"CloseFile"); 2
15 }
16
17 Instrument(int argc, char **argv, Obj *obj)
18 {
19    Proc *p; Block *b;
20
21      for (p = GetFirstObjProc(obj); p != NULL; p = GetNextProc(p)) { 3
22        AddCallProc(p,ProcBefore,"ProcedureCalls",n);
23        for (b = GetFirstBlock(p); b != NULL; b = GetNextBlock(b)) { 4
24          AddCallBlock(b,BlockBefore,"ProcedureCount", 5
25                       n,GetBlockInfo(b,BlockNumberInsts));
26        }
27        AddCallObj(obj, ObjAfter,"ProcedurePrint",n,SafeProcName(p)); 6
28        n++; 7
29      }
30 }
31
32 void InstrumentFini(void)
33 {
34   AddCallProgram(ProgramBefore,"OpenFile",n); 8
35 }
36
37 static const char *SafeProcName(Proc *p)
38 {
39     const char *        name;
40     static char         buf[128];
41
42     name = ProcName(p); 9
43     if (name)
44         return(name);
45     sprintf(buf, "proc_at_0x%lx", ProcPC(p));
46     return(buf);
47 }
```

1 Defines the interface to the analysis procedures.

2 Adds a call to the CloseFile analysis procedure to the end of the program.

3 Loops through each procedure in the object.

4 Loops through each basic block in the procedure.

5 Adds a call to the ProcedureCount analysis procedure before any of the instructions in this basic block are executed. The argument types of the ProcedureCount are defined in the prototype on line 11. The first argument is a procedure index of type int; the second argument, also an int, is the number of instructions in the basic block. The ProcedureCount analysis procedure adds the number of instructions in the basic block to a per-procedure data structure. Similarly, the ProcedureCalls analysis procedure increments a procedure's call count before each call begins executing the called procedure.

6 Adds a call to the `ProcedurePrint` analysis procedure to the end of the program. The `ProcedurePrint` analysis procedure prints a line summarizing this procedure’s instruction use and call count.

7 Increments the procedure index.

8 Adds a call to the `OpenFile` analysis procedure to the beginning of the program, passing it an `int` representing the number of procedures in the application. The `OpenFile` procedure allocates the per-procedure data structure that tallies instructions and opens the output file.

9 Determines the procedure name.

The analysis procedures used by the `iprof` tool are defined in the `iprof.anal.c` file as shown in the following example:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

long instrTotal = 0;
long *instrPerProc;
long *callsPerProc;

FILE *OpenUnique(char *fileName, char *type)
{
 FILE *file;
  char Name[200];

  if (getenv("ATOMUNIQUE") != NULL)
    sprintf(Name,"%s.%d",fileName,getpid());
  else
    strcpy(Name,fileName);

  file = fopen(Name,type);
  if (file == NULL)
    {
      fprintf(stderr,"Atom: can’t open %s for %s\n",Name, type);
      exit(1);
    }
  return(file);
}

static FILE *file;
void OpenFile(int number)
{
  file = OpenUnique("iprof.out","w");
  fprintf(file,"%30s %15s %15s %12s\n","Procedure","Calls",
          "Instructions","Percentage");
  instrPerProc = (long *) calloc(sizeof(long), number); 1
  callsPerProc = (long *) calloc(sizeof(long), number);
  if (instrPerProc == NULL || callsPerProc == NULL) {
    fprintf(stderr,"Malloc failed\n");
    exit(1);
  }
}

void ProcedureCalls(int number)
{
  callsPerProc[number]++;
}
```

```

void ProcedureCount(int number, int instructions)
{
  instrTotal += instructions;
  instrPerProc[number] += instructions;
}


void ProcedurePrint(int number, char *name)
{
  if (instrPerProc[number] > 0) { 2
    fprintf(file,"%30s %15ld %15ld %12.3f\n",
            name, callsPerProc[number], instrPerProc[number],
            100.0 * instrPerProc[number] / instrTotal);
  }
}

void CloseFile() 3
{
  fprintf(file,"\n%30s %15s %15ld\n", "Total", "", instrTotal);
  fclose(file);
}
```

1 Allocates the counts data structure. The `calloc` function zero-fills the counts data.

2 Filters procedures that are never called.

3 Closes the output file. Tools must explicitly close files that are opened in the analysis procedures.

After the instrumentation and analysis files are specified, the tool is complete. To demonstrate the application of this tool, compile and link the `"Hello"` application as follows:

```
#include <stdio.h>
main()
{
   printf("Hello world!\n");
}
```

The following example builds a call-shared executable, applies the `iprof` tool, and runs the instrumented executable. In contrast to the `ptrace` tool described in Section 9.2.8.1, the `iprof` tool sends its output to a file instead of `stdout`.

```
% cc hello.c -o hello
% atom hello iprof.inst.c iprof.anal.c -o hello.iprof -all
% setenv LD_LIBRARY_PATH ‘pwd‘
% hello.iprof
Hello world!
% more iprof.out
Procedure           Calls    Instructions   Percentage
__start               1              92        1.487
   main               1              15        0.242
   .
   .
   .
 printf               1              81        0.926
   .
   .
   .

  Total                            8750
% unsetenv LD_LIBRARY_PATH
```

9.2.8.3 Data Cache Simulation Tool

Instruction and data address tracing has been used for many years as a technique to capture and analyze cache behavior. Unfortunately, current machine speeds make this increasingly difficult. For example, the Alvinn SPEC92 benchmark executes 961,082,150 loads, 260,196,942 stores, and 73,687,356 basic blocks, for a total of 2,603,010,614 Alpha instructions. Storing the address of each basic block and the effective address of all the loads and stores would take in excess of 10 GB and slow down the application by a factor of over 100.

The `cache` tool uses on-the-fly simulation to determine the cache miss rates of an application running in an 8-KB, direct-mapped cache. The following example shows its instrumentation routine:

```
#include <stdio.h>
#include <cmplrs/atom.inst.h>

unsigned InstrumentAll(int argc, char **argv)
{
  Obj *o; Proc *p;  Block *b;  Inst *i;

  AddCallProto("Reference(VALUE)");
  AddCallProto("Print()");
  for (o = GetFirstObj(); o != NULL; o = GetNextObj(o)) {
    if (BuildObj(o)) return (1);
    for (p=GetFirstProc(); p != NULL; p = GetNextProc(p)) {
      for (b = GetFirstBlock(p); b != NULL;  b = GetNextBlock(b)) {
        for (i = GetFirstInst(b); i != NULL; i = GetNextInst(i)) { 1
          if (IsInstType(i,InstTypeLoad) || IsInstType(i,InstTypeStore)) {
            AddCallInst(i,InstBefore,"Reference",EffAddrValue); 2
          }
        }
      }
    }
    WriteObj(o);
  }
AddCallProgram(ProgramAfter,"Print");
return (0);
}
```

1 Examines each instruction in the current basic block.

2 If the instruction is a load or a store, adds a call to the `Reference` analysis procedure, passing the effective address of the data reference.

The analysis procedures used by the `cache` tool are defined in the `cache.anal.c` file as shown in the following example:

```
#include <stdio.h>
#include <assert.h>
#define CACHE_SIZE 8192
#define BLOCK_SHIFT 5
long tags[CACHE_SIZE >> BLOCK_SHIFT];
long references, misses;

void Reference(long address) {
  int index = (address & (CACHE_SIZE-1)) >> BLOCK_SHIFT;
  long tag = address >> BLOCK_SHIFT;
  if tags[index] != tag) {
    misses++;
    tags[index] = tag;
  }
  references++;
}
void Print() {
  FILE *file = fopen("cache.out","w");
  assert(file != NULL);
  fprintf(file,"References: %ld\n", references);
  fprintf(file,"Cache Misses: %ld\n", misses);
  fprintf(file,"Cache Miss Rate: %f\n", (100.0 * misses) / references);
  fclose(file);
}
```

After the instrumentation and analysis files are specified, the tool is complete. To demonstrate the application of this tool, compile and link the `"Hello"` application as follows:

```
#include <stdio.h>
main()
{
   printf("Hello world!\n");
}
```

The following example applies the `cache` tool to instrument both the nonshared and call-shared versions of the application:

```
% cc hello.c -o hello
% atom hello cache.inst.c cache.anal.c -o hello.cache -all
% setenv LD_LIBRARY_PATH ‘pwd‘
% hello.cache
Hello world!
% more cache.out
References: 1091
Cache Misses: 225
Cache Miss Rate: 20.623281
% cc -non_shared hello.c -o hello
% atom hello cache.inst.c cache.anal.c -o hello.cache -all
% hello.cache
Hello world!
```

```
% more cache.out
References: 382
Cache Misses: 93
Cache Miss Rate: 24.345550
```

10

Optimizing Techniques

Optimizing an application program can involve modifying the build process, modifying the source code, or both.

In many instances, optimizing an application program can result in major improvements in run-time performance. Two preconditions should be met, however, before you begin measuring the run-time performance of an application program and analyzing how to improve the performance:

- Check the software on your system to ensure that you are using the latest versions of the compiler and the operating system to build your application program. Newer versions of a compiler often perform more advanced optimizations, and newer versions of the operating system often operate more efficiently.
- Test your application program to ensure that it runs without errors. Whether you are porting an application from a 32-bit system to Tru64 UNIX or developing a new application, never attempt to optimize an application until it has been thoroughly debugged and tested. (If you are porting an application written in C, compile your program using the C compiler's `-message_enable questcode` option, and/or use `lint` with the `–Q` option to help identify possible portability problems that you may need to resolve.)

After you verify that these conditions have been met, you can begin the optimization process.

The process of optimizing an application can be divided into two separate, but complementary, activities:

- Tuning your application's build process so that you use, for example, an optimal set of automatic preprocessing and compilation optimizations (see Section 10.1).
- Analyzing your application's source code to ensure that it uses efficient algorithms, and that it does not use programming language constructs that can degrade performance (see Section 10.2). This manual phase also includes the use of profiling tools to analyze performance, as explained in Chapter 8.

The following sections provide details that relate to these two aspects of the optimization process.

10.1 Guidelines to Build an Application Program

Opportunities to automatically improve an application's run-time performance exist in all phases of the build process. The following sections identify some of the major opportunities that exist in the areas of compiling, linking and loading, preprocessing and postprocessing, and library selection. A particularly effective technique is profile-directed optimization with the `spike` tool (Section 10.1.3).

10.1.1 Compilation Considerations

Compile your application with the highest optimization level possible, that is, the level that produces the best performance and the correct results. In general, applications that conform to language-usage standards should tolerate the highest optimization levels, and applications that do not conform to such standards may have to be built at lower optimization levels. See `cc`(1) or Chapter 2 for more information.

If your application will tolerate it, compile all of the source files together in a single compilation. Compiling multiple source files increases the amount of code that the compiler can examine for possible optimizations. This can have the following effects:

- More procedure inlining
- More complete data flow analysis
- A reduction in the number of external references to be resolved during linking

To take advantage of these optimizations, use the `-ifo` and either `–O3` or `–O4` compilation options.

To determine whether the highest level of optimization benefits your particular program, compare the results of two separate compilations of the program, with one compilation at the highest level of optimization and the other compilation at the next lower level of optimization. Some routines may not tolerate a high level of optimization; such routines will have to be compiled separately.

Other compilation considerations that can have a significant impact on run-time performance include the following:

- For C applications with numerous floating-point operations, consider using the `–fp_reorder` option if a small difference in the result is acceptable.
- If your C application uses a lot of `char`, `short`, or `int` data items within loops, you may be able to use the C compiler's highest-level optimization option to improve performance. (The highest-level optimization option

(–O4) implements byte vectorization, among other optimizations, for Alpha systems.)

- For C and Fortran applications whose performance can be characterized using one or more sample runs, consider using the `-feedback` option. This option is especially effective when used with the `-spike` option, as discussed in Section 10.1.3.2, or the `-ifo` option for even better results.
- For C applications that are thoroughly debugged and that do not generate any exceptions, consider using the `–speculate` option. When a program compiled with this option is executed, values associated with a variety of execution paths are precomputed so that they are immediately available if they are needed. This work ahead operation uses idle machine cycles, so it has no negative effect on performance. Performance is usually improved whenever a precomputed value is used.

  The `–speculate` option can be specified in two forms:

  ```
  –speculate all
  –speculate by_routine
  ```

  Both options result in exceptions being dismissed: the `–speculate all` option dismisses exceptions generated in all compilation units of the program, and the `–speculate by_routine` option dismisses only the exceptions in the compilation unit to which it applies. If speculative execution results in a significant number of dismissed exceptions, performance will be degraded. The `–speculate all` option is more aggressive and may result in greater performance improvements than the other option, especially for programs doing floating-point computations. The `–speculate all` option cannot be used if any routine in the program does exception handling; however, the `–speculate by_routine` option can be used when exception handling occurs outside the compilation unit on which it is used. Neither `–speculate` option should be used if debugging is being done.

  To print a count of the number of dismissed exceptions when the program does a normal termination, specify the following environment variable:

  ```
  % setenv _SPECULATE_ARGS -stats
  ```

  The statistics feature is not currently available with the `–speculate all` option.

  Use of the `–speculate all` and `–speculate by_routine` options disables all messages about alignment fixups. To generate alignment messages for both speculative and nonspeculative alignment fixups, specify the following environment variable:

  ```
  % setenv _SPECULATE_ARGS -alignmsg
  ```

  You can specify both options as follows:

```
% setenv _SPECULATE_ARGS -stats -alignmsg
```

- You can use the following compilation options together or individually to improve run-time performance (see cc(1) for more information):

Option	Description
-arch	Specifies which version of the Alpha architecture to generate instructions for. See -arch in cc(1) for an explanation of the differences between -arch and -tune.
–ansi_alias	Specifies whether source code observes ANSI C aliasing rules. ANSI C aliasing rules allow for more aggressive optimizations.
–ansi_args	Specifies whether source code observes ANSI C rules about arguments. If ANSI C rules are observed, special argument-cleaning code does not have to be generated.
–fast	Turns on the optimizations for the following options for increased performance: –ansi_alias –ansi_args –assume trusted_short_alignment –D_FASTMATH –float –fp_reorder –ifo –D_INLINE_INTRINSICS –D_INTRINSICS -intrinsics –O3 –readonly_strings
–feedback	Specifies that the compiler should use the profile information contained in the specified file when performing optimizations. For more information, see Section 10.1.3.2.
–fp_reorder	Specifies whether certain code transformations that affect floating-point operations are allowed.
–G	Specifies the maximum byte size of data items in the small data sections (sbss or sdata).
–inline	Specifies whether to perform inline expansion of functions.
–ifo	Provides improved optimization (interfile optimization) and code generation across file boundaries that would not be possible if the files were compiled separately.
–O	Specifies the level of optimization that is to be achieved by the compilation.

Option	Description
–om	Performs a variety of postlink code optimizations. Most effective with programs compiled with the –non_shared option (see Appendix F). This option is being replaced with the -spike option (see Section 10.1.3).
–preempt_module	Supports symbol preemption on a module-by-module basis.
–speculate	Enables work (for example, load or computation operations) to be done in running programs on execution paths before the paths are taken.
–spike	Performs a variety of postlink code optimizations (see Section 10.1.3).
–tune	Selects processor-specific instruction tuning for specific implementations of the Alpha architecture. See -arch in cc(1) for an explanation of the differences between -tune and -arch.
–unroll	Controls loop unrolling done by the optimizer at levels –O2 and above.

Using the preceding options may cause a reduction in accuracy and adherence to standards.

- For C applications, the compilation option in effect for handling floating-point exceptions can have a significant impact on execution time as follows:
  - **Default exception handling** (no special compilation option)

    With the default exception-handling mode, overflow, divide-by-zero, and invalid-operation exceptions always signal the SIGFPE exception handler. Also, any use of an IEEE infinity, an IEEE NaN (not-a-number), or an IEEE denormalized number will signal the SIGFPE exception handler. By default, underflows silently produce a zero result, although the compilers support a separate option that allows underflows to signal the SIGFPE handler.

    The default exception-handling mode is suitable for any portable program that does not depend on the special characteristics of particular floating-point formats. The default mode provides the best exception-handling performance.
  - **Portable IEEE exception handling** (–ieee)

    With the portable IEEE exception-handling mode, floating-point exceptions do not signal unless a special call is made to enable the fault. This mode correctly produces and handles IEEE infinity, IEEE NaNs, and IEEE denormalized numbers. This mode also provides support for most of the nonportable aspects of IEEE floating point: all status options and trap enables are supported, except for the inexact exception. (See ieee(3) for information on the inexact exception feature (–ieee_with_inexact). Using this feature can slow down floating-point calculations by a factor of 100 or more, and few, if any, programs have a need for its use.)

The portable IEEE exception-handling mode is suitable for any program that depends on the portable aspects of the IEEE floating-point standard. This mode is usually 10-20 percent slower than the default mode, depending on the amount of floating-point computation in the program. In some situations, this mode can increase execution time by more than a factor of two.

10.1.2 Linking and Loading Considerations

If your application does not use many large libraries, consider linking it nonshared. This allows the linker to optimize calls into the library, which decreases your application's startup time and improves run-time performance (if calls are made frequently). Nonshared applications, however, can use more system resources than call-shared applications. If you are running a large number of applications simultaneously and the applications have a set of libraries in common (for example, libX11 or libc), you may increase total system performance by linking them as call-shared. See Chapter 4 for details.

For applications that use shared libraries, ensure that those libraries can be quickstarted. Quickstarting is a Tru64 UNIX capability that can greatly reduce an application's load time. For many applications, load time is a significant percentage of the total time that it takes to start and run the application. If an object cannot be quickstarted, it still runs, but startup time is slower. See Section 4.7 for details.

10.1.3 Spike and Profile-Directed Optimization

This section describes use of the spike postlink optimizer.

10.1.3.1 Overview of spike

The spike tool performs code optimization after linking. Because it can operate on an entire program, spike is able to do optimizations that the compiler cannot do. spike is most effective when it uses profile information to guide optimization, as discussed in Section 10.1.3.2.

spike is new with Tru64 UNIX Version 5.1 and is intended to replace om and cord. It provides better control and more effective optimization, and it can be used with both executables and shared libraries. spike cannot be used with om or cord. For information about om and cord, see Appendix F.

Some of the optimizations that spike performs are code layout, deleting unreachable code, and optimization of address computations.

spike can process binaries that are linked on Tru64 UNIX V4.0 or later systems. Binaries that are linked on V5.1 or later systems contain information that allows spike to do additional optimization.

Note

> spike does only some address optimizations on Tru64 UNIX V5.1 or later images, but om will do the optimization on V4 images. If you are using spike on pre V5.1 binaries and you enable linker optimization (-O passed to cc in the link step), the difference in performance between om and spike is not expected to be significant.

You can use spike in two ways:

- By applying the spike command to a binary file after compilation.
- As part of the compilation process, by specifying the -spike option with the cc command (or with the cxx, f77, or f90 command, if the corresponding compiler is installed on your system).

The examples in this section and Section 10.1.3.2 show how to use both forms of spike The spike command is more convenient when you do not want to relink the executable (Example 1) or when you are using profile information after compilation (Example 5 and Example 6). The -spike option is more convenient when you are not using profile information (Example 2), or when you are using profile information in the compiler, too (Example 3 and Example 4).

Example 1 and Example 2 show how to use spike without profiling information to guide the optimization. Section 10.1.3.2 explains how to use spike with feedback information from the pixie profiler.

Example 1

In this example, spike is applied to the binary my_prog, producing the optimized output file prog1.opt.

```
% spike my_prog -o prog1.opt
```

Example 2

In this example, spike is applied during compilation with the cc command's -spike option:

```
% cc -c file1.c
% cc -o prog3 file1.o -spike
```

The first command line creates the object file file1.o. The second command line links file1.o into an executable and uses spike to optimize the executable.

All of the spike command's options can be passed directly to the cc command's -spike option by using the (cc) -WS option. The following example shows the syntax:

```
% cc -spike -feedback prog -o prog *.c \
     -WS,-splitThresh,.999,-noaggressiveAlign
```

For complete information on the spike command's options and any restrictions on using spike, see spike(1).

10.1.3.2 Using spike for Profile-Directed Optimization

You can achieve some degree of automatic optimization by using the compiler's automatic optimization options that are described in the previous sections, such as -O, -fast, -inline, and so on. These options can help in the generation of minimal instruction sequences that make best use of the CPU architecture and cache memory.

However, the compiler and linker can improve on these optimizations if given information on which instructions are executed most often when a program is run with its normal input data and environment. Tru64 UNIX helps you provide this information by allowing a profiler's results to be fed back into a recompilation. This customized, profile-directed optimization can be used in conjunction with automatic optimization.

The following examples show how to use spike with the pixie profiler and various feedback techniques to tune the generated instruction sequences of a program.

Example 3

This example shows the three basic steps for profile-directed optimization with spike: (1) preparing the program for optimization, (2) creating an instrumented version of the program and running it to collect profiling statistics, and (3) feeding that information back to the compiler and linker to help them optimize the executable code. Later examples show how to elaborate on these steps to accommodate ongoing changes during development and data from multiple profiling runs.

```
% cc -feedback prog -o prog -O3 *.c 1
% pixie -update prog 2
% cc -feedback prog -o prog -spike -O3 *.c 3
```

1 When the program is compiled with the -feedback option for the first time, a special augmented executable file is created. It contains

information that the compiler uses to relate the executable to the source files. It also contains a section that is used later to store profiling feedback information for the compiler. This section remains empty after the first compilation because the pixie profiler has not yet generated any feedback information (step 2). Make sure that the file name specified with the -feedback option is the same as the executable file name, which in this example is prog (from the -o option). By default, the -feedback option applies the -g1 option, which provides optimum symbolization for profiling. You need to experiment with the -O*n* option to find the level of optimization that provides the best run-time performance for your program and compiler. The compiler issues this message during the first compilation, because no feedback information is yet available:

```
cc: Info: Feedback file prog does not exist (nofbfil)
cc: Info: Compilation will proceed without feedback optimizations (nofbopt)
```

2 The pixie command creates an instrumented version of the program (prog.pixie) and then runs it (because a prof option, -update, is specified). Execution statistics and address mapping data are automatically collected in an instruction-counts file (prog.Counts) and an instruction-addresses file (prog.Addrs). The -update option puts this profiling information in the augmented executable.

3 In the second compilation with the -feedback option, the profiling information in the augmented executable guides the compiler and (through the -spike option) the postlink optimizer. This customized feedback enhances any automatic optimization that the -O3 and -spike options provide. You can make compiler optimizations even more effective by using the -ifo and/or -assume whole_program options in conjunction with the -feedback option. However, as noted in Section 10.1.1, the compiler may be unable to compile very large programs as if there were only one source file.

See pixie(1) and cc(1) for more information.

The profiling information in an augmented executable file makes it larger than a normal executable (typically 3–5 percent). After development is completed, you can use the strip command to remove any profiling and symbol table information. For example:

```
% strip prog
```

spike cannot process stripped images.

Example 4

During a typical development process, steps 2 and 3 of Example 3 are repeated as needed to reflect the impact of any changes to the source code. For example:

```
% cc -feedback prog -o prog -O3 *.c
% pixie -update prog
% cc -feedback prog -o prog -O3 *.c
[modify source code]
% cc -feedback prog -o prog -O3 *.c
.....
[modify source code]
% cc -feedback prog -o prog -O3 *.c
% pixie -update prog
% cc -feedback prog -o prog -spike -O3 *.c
```

Because the profiling information in the augmented executable persists from compilation to compilation, the `pixie` processing step that updates the information does not have to be repeated every time that a source module is modified and recompiled. But each modification reduces the relevance of the old feedback information to the actual code and degrades the potential quality of the optimization, depending on the exact modification. The `pixie` processing step after the last modification and recompilation guarantees that the feedback information is correctly updated for the last compilation.

Example 5

You might want to run your instrumented program several times with different inputs to get an accurate picture of its profile. This example shows how to optimize a program by merging profiling statistics from two instrumented runs of a program, `prog`, whose output varies from run to run with different sets of input:

```
% cc -feedback prog -o prog *.c 1
% pixie -pids prog 2
% prog.pixie 3
(input set 1)
% prog.pixie
(input set 2)
% prof -pixie -update prog prog.Counts.* 4
% spike prog -feedback prog -o prog.opt 5
```

1 The first compilation produces an augmented executable, as explained in Example 3.

2 By default, each run of the instrumented program (`prog.pixie`) produces a profiling data file called `prog.Counts`. The `-pids` option adds the process ID of each of the instrumented program's test runs to the name of the profiling data file that is produced (`prog.Counts.`*`pid`*). Thus, the data files that subsequent runs produce do not overwrite each other.

3 The instrumented program is run twice, producing a uniquely named data file each time — for example, prog.Counts.371 and prog.Counts.422.

4 The prof -pixie command merges the two data files. The -update option updates the executable, prog, with the combined information.

5 The spike command with the -feedback option uses the combined profiling information from the two runs of the program to guide the optimization, producing the optimized output file prog.opt.

The last step of this example could be changed to the following:

```
% cc -spike -feedback prog -o prog -O3 *.c
```

The -spike option requires that you relink the program. When using the spike command, you do not have to link the program a second time to invoke spike.

Example 6

This example differs from Example 5 in that a normal (unaugmented) executable is created, and the spike command’s -fb option (rather than the -feedback option) is used:

```
% cc prog -o prog *.c
% pixie -pids prog
% prog.pixie
(input set 1)
% prog.pixie
(input set 2)
% prof -pixie -merge prog.Counts prog prog.Addrs prog.Counts.*
% spike prog -fb prog -o prog.opt
```

The prof -pixie -merge command merges the two data files from the two instrumented runs into one combined prog.Counts file. With this form of feedback, the -g1 option must be specified explicitly to provide optimum symbolization for profiling.

The spike -fb command uses the information in prog.Addrs and prog.Counts to produce the optimized output file prog.opt.

The method of Example 5 is preferred. The method in Example 6 is supported for compatibility and should be used only if you cannot compile with the -feedback option that uses feedback information stored in the executable.

10.1.4 Preprocessing and Postprocessing Considerations

Preprocessing options and postprocessing (run-time) options that can affect performance include the following:

- Use the Kuck & Associates Preprocessor (KAP) tool to gain extra optimizations. The preprocessor uses final source code as input and produces an optimized version of the source code as output.

  KAP is especially useful for applications with the following characteristics on both symmetric multiprocessing systems (SMP) and uniprocessor systems:

  – Programs with a large number of loops or loops with large loop bounds

  – Programs that act on large data sets

  – Programs with significant reuse of data

  – Programs with a large number of procedure calls

  – Programs with a large number of floating-point operations

  To take advantage of the parallel-processing capabilities of SMP systems, the KAP preprocessors support automatic and directed decomposition for C programs. KAP's automatic decomposition feature analyzes an existing program to locate loops that are candidates for parallel execution. Then, it decomposes the loops and inserts all necessary synchronization points. If more control is desired, the programmer can manually insert directives to control the parallelization of individual loops. On Tru64 UNIX systems, KAP uses the POSIX Threads Library to implement parallel processing.

  For C programs, KAP is invoked with the `kapc` (which invokes separate KAP processing) or `kcc` command (which invokes combined KAP processing and HP C compilation). For information on how to use KAP on a C program, see the *KAP for C for Tru64 UNIX* manual.

  KAP is available for Tru64 UNIX systems as a separately orderable layered product.

- Use the following tools, especially with profile-directed feedback, for post-link optimization and procedure reordering:

  – `spike` (see Section 10.1.3)

  – `om` (see Appendix F)

  – `cord` (see Appendix F)

10.1.5 Library Routine Selection

Library routine options that can affect performance include the following:

- Use the Compaq Extended Math Library (CXML, formerly Digital Extended Math Library — DXML) for applications that perform numerically intensive operations. CXML is a collection of mathematical routines that are optimized for Alpha systems — both SMP systems and uniprocessor systems. The routines in CXML are organized in the following four libraries:

  - BLAS — A library of basic linear algebra subroutines
  - LAPACK — A linear algebra package of linear system and eigensystem problem solvers
  - Sparse Linear System Solvers — A library of direct and iterative sparse solvers
  - Signal Processing — A basic set of signal-processing functions, including one-, two-, and three-dimensional fast Fourier transforms (FFTs), group FFTs, sine/cosine transforms, convolution functions, correlation functions, and digital filters

  By using CXML, applications that involve numerically intensive operations may run significantly faster on Tru64 UNIX systems, especially when used with KAP. CXML routines can be called explicitly from your program or, in certain cases, from KAP (that is, when KAP recognizes opportunities to use the CXML routines). You access CXML by specifying the `–ldxml` option on the compilation command line.

  For details on CXML, see the *Compaq Extended Math Library Reference Guide*.

  The CXML routines are written in Fortran. For information on calling Fortran routines from a C program, see the Compaq Fortran (formerly Digital Fortran) user manual for Tru64 UNIX. (Information about calling CXML routines from C programs is also provided in the *TechAdvantage C/C++ Getting Started Guide*.)

- If your application does not require extended-precision accuracy, you can use math library routines that are faster but slightly less accurate. Specifying the `–D_FASTMATH` option on the compilation command causes the compiler to use faster floating-point routines at the expense of three bits of floating-point accuracy. See `cc`(1) for more information.

- Consider compiling your C programs with the `–D_INTRINSICS` and `–D_INLINE_INTRINSICS` options; this causes the compiler to inline calls to certain standard C library routines.

10.2 Application Coding Guidelines

If you are willing to modify your application, use the profiling tools to determine where your application spends most of its time. Many applications spend most of their time in a few routines. Concentrate your efforts on improving the speed of those heavily used routines.

Tru64 UNIX provides several profiling tools that work for programs written in C and other languages. See Chapter 7, Chapter 8, Chapter 9, `prof_intro`(1), `gprof`(1), `hiprof`(1), `pixie`(1), `prof`(1), `third`(1), `uprofile`(1), and `atom`(1) for more information.

After you identify the heavily used portions of your application, consider the algorithms used by that code. Is it possible to replace a slow algorithm with a more efficient one? Replacing a slow algorithm with a faster one often produces a larger performance gain than tweaking an existing algorithm.

When you are satisfied with the efficiency of your algorithms, consider making code changes to help the compiler optimize the object code that it generates for your application. *High Performance Computing* by Kevin Dowd (O'Reilly & Associates, Inc., ISBN 1-56592-032-5) is a good source of general information on how to write source code that maximizes optimization opportunities for compilers.

The following sections identify performance opportunities involving data types, I/O handling, cache usage and data alignment, and general coding issues.

10.2.1 Data-Type Considerations

Data-type considerations that can affect performance include the following:

- The smallest unit of efficient access on Alpha systems is 32 bits. A 32- or 64-bit data item can be accessed with a single, efficient machine instruction.If your application's performance on older implementations of the Alpha architecture (processors earlier than EV56) is critical, you may want to consider the following points:
  - Avoid using integer and logical data types that are less than 32 bits, especially for scalars that are used frequently.
  - In C programs, consider replacing `char` and `short` declarations with `int` and `long` declarations.
- Division of integer quantities is slower than division of floating-point quantities. If possible, consider replacing such integer operations with equivalent floating-point operations.

  Integer division operations are not native to the Alpha processor and must be emulated in software, so they can be slow. Other non-native operations include transcendental operations (for example, sine and cosine) and square root.

10.2.2 Using Direct I/O on AdvFS Files

Direct I/O allows an application to use the file-system features that the Advanced File System (AdvFS) provides, such as file management, online backup, and online recovery, while eliminating the overhead of copying user data into the AdvFS cache. Direct I/O uses Direct Memory Access (DMA) commands to copy the user data directly between an application's buffer and a disk.

Normal file-system I/O maintains file pages in a cache. This allows the I/O to be completed asynchronously; once the data is in the cache and scheduled for I/O, the application does not need to wait for the data to be transferred to disk. In addition, because the data is already in the cache, subsequent accesses to this page do not need to read the data from disk. Most applications use normal file-system I/O.

Normal file-system I/O is not suited for applications that access the data on disk infrequently and manage inter-thread competition themselves. Such applications can take advantage of the reduced overhead of direct I/O. However, because data is not cached, access to a given page must be serialized among competing threads. To do this, direct I/O enforces synchronous I/O as the default. This means that when the `read( )` routine returns to the application, the I/O has completed and the data is on disk. Any subsequent retrieval of that data will also incur an I/O operation to retrieve the data from disk.

An application can take advantage of asynchronous I/O (AIO), but still use the underlying direct I/O mechanism, by using the `aio_read( )` and `aio_write( )` system routines. These routines will return to the application before the data has been transferred to disk, and the `aio_error( )` routine allows the application to poll for the completion of the I/O. (The kernel synchronizes the access to file pages so that two threads cannot concurrently write the same page.)

Threads using direct I/O to access a given file will be able to do so concurrently, provided that they do not access the same range of pages. For example, if thread A is writing pages 10 through 19 and thread B is writing pages 20 through 39, these operations will occur simultaneously. Continuing this example, if thread B attempts to write pages 15 through 39 in a single direct I/O transfer, it will be forced to wait until thread A completes its write because their page ranges overlap.

When using direct I/O, the best performance occurs when the requested transfer is aligned on a disk sector boundary and the transfer size is an even multiple of the underlying sector size. Larger transfers are generally more efficient than smaller ones, although the optimal transfer size depends on the underlying storage hardware.

Note

Direct I/O mode and the use of mapped file regions (`mmap`) are exclusive operations. You cannot set direct I/O mode on a file that uses mapped file regions. Mapping a file will also fail if the file is already open for direct I/O.

Direct I/O and atomic data logging modes are also mutually exclusive. If a file is open in one of these modes, subsequent attempts to open the file in the other mode will fail.

You can activate the direct I/O feature for use on an AdvFS file for both AIO and non-AIO applications. To activate the feature, use the `open` function in an application, setting the `O_DIRECTIO` file access flag. For example:

```
 open ("file", O_DIRECTIO | O_RDWR, 0644)
```

Direct I/O mode remains in effect until the file is closed by all users.

The `fcntl( )` function with the parameter `F_GETCACHEPOLICY` can be used to return the caching policy of a file, either `FCACHE` or `FDIRECTIO` mode. For example:

```
int fcntlarg = 0;
ret = fcntl( filedescriptor, F_GETCACHEPOLICY, &fcntlarg );
if ( ret != -1 && fcntlarg == FDIRECTIO ) {
.
.
.
```

For details on the use of direct I/O and AdvFS, see `fcntl`(2) and `open`(2).

10.2.3 Cache Usage and Data Alignment Considerations

Cache usage patterns can have a critical impact on performance:

- If your application has a few heavily used data structures, try to allocate these data structures on cache-line boundaries in the secondary cache. Doing so can improve the efficiency of your application's use of cache. See the*Alpha Architecture Reference Manual* for more information.
- Look for potential data cache collisions between heavily used data structures. Such collisions occur when the distance between two data structures allocated in memory is equal to the size of the primary (internal) data cache. If your data structures are small, you can avoid this by allocating them contiguously in memory. You can use the `uprofile` tool to determine the number of cache collisions and their locations. See the *Alpha Architecture Reference Manual* for more information on data cache collisions.

Data alignment can also affect performance. By default, the C compiler aligns each data item on its natural boundary; that is, it positions each data item so that its starting address is an even multiple of the size of the data type used to declare it. Data not aligned on natural boundaries is called misaligned data. Misaligned data can slow performance because it forces the software to make necessary adjustments at run time.

In C programs, misalignment can occur when you type cast a pointer variable from one data type to a larger data type; for example, type casting a `char` pointer (1-byte alignment) to an `int` pointer (4-byte alignment) and then dereferencing the new pointer may cause unaligned access. Also in C, creating packed structures using the `#pragma pack` directive can cause unaligned access. (See Chapter 3 for details on the `#pragma pack` directive.)

To correct alignment problems in C programs, you can use the `-misalign` option (or `-assume noaligned_objects`) or you can make necessary modifications to the source code. If instances of misalignment are required by your program for some reason, use the `__unaligned` data-type qualifier in any pointer definitions that involve the misaligned data. When data is accessed through the use of a pointer declared `__unaligned`, the compiler generates the additional code necessary to copy or store the data without generating alignment errors. (Alignment errors have a much more costly impact on performance than the additional code that is generated.)

Warning messages identifying misaligned data are not issued during the compilation of C programs. However, during execution of any program, the kernel issues warning messages (“unaligned access”) for most instances of misaligned data. The messages include the program counter (PC) value for the address of the instruction that caused the misalignment.

You can use either of the following two methods to access code that causes the unaligned access fault:

- By using a debugger to examine the PC value presented in the "unaligned access" message, you can find the routine name and line number for the instruction causing the misalignment. (In some cases, the "unaligned access" message results from a pointer passed by a calling routine. The return address register (ra) contains the address of the calling routine — if the contents of the register have not been changed by the called routine.)
- By turning off the `-align` option on the command line and running your program in a debugger session, you can examine your program’s stack and variables at the point where the debugger stops due to the unaligned access.

For more information on data alignment, see the *Alpha Architecture Reference Manual*. See `cc`(1) for information on alignment-control options that you can specify on compilation command lines.

10.2.4 General Coding Considerations

General coding considerations specific to C applications include the following:

- Use libc functions (for example: strcpy, strlen, strcmp, bcopy, bzero, memset, memcpy) instead of writing similar routines or your own loops. These functions are hand coded for efficiency.
- Use the unsigned data type for variables wherever possible because:
  - The variable is always greater than or equal to zero, which enables the compiler to perform optimizations that would not otherwise be possible.
  - The compiler generates fewer instructions for all unsigned divide operations.

  Consider the following example:

  ```
  int long i;
  unsigned long j;
     .
     .
  return i/2 + j/2;
  ```

  In the example, i/2 is an expensive expression; however, j/2 is inexpensive.

  The compiler generates three instructions for the signed i/2 operations:

  ```
  addq $l, l, $28
  cmovge $l, $l, $28
  sra $28, l, $2
  ```

  The compiler generates only one instruction for the unsigned j/2 operation:

  ```
  srl $3, 1, $4
  ```

  Also, consider using the -unsigned option to treat all char declarations as unsigned char.
- If your application temporarily needs large amounts of data, consider using the malloc function or the alloca built-in function instead of declaring the data statically.

  The alloca function allocates memory from the stack. The memory is automatically released when the function that allocated it returns. You must make sure that any code that uses alloca first includes alloca.h. If you do not do this, your code may not work correctly.

  Consider using the malloc function if your application needs memory to live beyond the context of a specific function invocation. The malloc function allocates memory from the process's heap. The memory remains available until it is explicitly released by a call to free.

  Using these functions can increase the performance of applications where physical memory is a scarce resource.

For multithreaded applications, note that alloca allocates memory from the calling thread's stack, which means that the allocating and freeing of this memory does not incur any contention. The malloc function (and associated functions) may allocate their memory from a common pool using locking and atomic operations to control concurrent access. See the Tuning Memory Allocation section of malloc(3) for information on simple ways to improve the performance of single and multithreaded applications that use malloc.

Also for multithreaded applications, consider using the arena malloc (amalloc) mechanism to set up separate heaps for each thread of a multithreaded application.

- Minimize type casting, especially type conversion from integer to floating point and from a small data type to a larger data type.
- To avoid cache misses, make sure that multidimensional arrays are traversed in natural storage order; that is, in row major order with the rightmost subscript varying fastest and striding by 1. Avoid column major order (which is used by Fortran).
- If your application fits in a 32-bit address space and allocates large amounts of dynamic memory by allocating structures that contain many pointers, you may be able to save significant amounts of memory by using the -xtaso option. To use this option, you must modify your source code with a C-language pragma that controls pointer size allocations. See cc(1) and Chapter 2 for information.
- Do not use indirect calls in C programs (that is, calls that use routines or pointers to functions as arguments). Indirect calls introduce the possibility of changes to global variables. This effect reduces the amount of optimization that can be safely performed by the optimizer.
- Use functions to return values instead of reference parameters.
- Use do while instead of while or for whenever possible. With do while, the optimizer does not have to duplicate the loop condition to move code from within the loop to outside the loop.
- Use local variables and avoid global variables. Declare any variable outside a function as static, unless that variable is referenced by another source file. Minimizing the use of global variables increases optimization opportunities for the compiler.
- Use value parameters instead of reference parameters or global variables. Reference parameters have the same degrading effects as pointers.
- Write straightforward code. For example, do not use ++ and -- operators within an expression. When you use these operators for their values

instead of their side effects, you often get bad code. For example, the following coding is not recommended:

```
while (n--)
  {
   .
   .
  }
```

The following coding is recommended:

```
while (n != 0)
  {
  n--;
   .
   .
  }
```

- Avoid taking and passing addresses (that is, & values). Using & values can create aliases, make the optimizer store variables from registers to their home storage locations, and significantly reduce optimization opportunities.
- Avoid creating functions that take a variable number of arguments. A function with a variable number of arguments causes the optimizer to unnecessarily save all parameter registers on entry.
- Declare functions as `static` unless the function is referenced by another source module. Use of `static` functions allows the optimizer to use more efficient calling sequences.

Also, avoid aliases where possible by introducing local variables to store dereferenced results. (A dereferenced result is the value obtained from a specified address.) Dereferenced values are affected by indirect operations and calls, but local variables are not; local variables can be kept in registers. Example 10–1 shows how the proper placement of pointers and the elimination of aliasing enable the compiler to produce better code.

Example 10–1: Pointers and Optimization

```
Source Code:
int len = 10;
char a[10];

void zero()
  {
  char *p;
  for (p = a; p != a + len; ) *p++ = 0;
  }
```

Consider the use of pointers in Example 10–1. Because the statement `*p++ = 0` might modify `len`, the compiler must load it from memory and add it to the address of `a` on each pass through the loop, instead of computing `a + len` in a register once outside the loop.

You can use two different methods to increase the efficiency of the code used in Example 10–1:

- Use subscripts instead of pointers. As shown in the following example, the use of subscripting in the `azero` procedure eliminates aliasing; the compiler keeps the value of `len` in a register, saving two instructions, and still uses a pointer to access `a` efficiently, even though a pointer is not specified in the source code:

  ```
  Source Code:
  char a[10];
  int len;

  void azero()
    {
    int i;
    for (i = 0; i != len; i++) a[i] = 0;
    }
  ```

- Use local variables. As shown in the following example, specifying `len` as a local variable or formal argument ensures that aliasing cannot take place and permits the compiler to place `len` in a register:

  ```
  Source Code:
       char a[10];
       void lpzero(len)
         int len;
         {
         char *p;
         for (p = a; p != a + len; ) *p++ = 0;
         }
  ```

11

Handling Exception Conditions

An exception is a special condition that occurs during the currently executing thread and requires the execution of code that acknowledges the condition and performs some appropriate actions. This code is known as an exception handler.

A termination handler consists of code that executes when the flow of control leaves a specific body of code. Termination handlers are useful for cleaning up the context established by the exiting body of code, performing such tasks as freeing memory buffers or releasing locks.

This chapter covers the following topics:

- Overview of exception handling (Section 11.1)
- Raising an exception from a user program (Section 11.2)
- Writing a structured exception handler (Section 11.3)
- Writing a termination handler (Section 11.4)

11.1 Exception-Handling Overview

On Tru64 UNIX systems, hardware traps exceptions, as described in the *Alpha Architecture Reference Manual*, and delivers them to the operating system kernel. The kernel converts certain hardware exceptions, such as bad memory accesses and arithmetic traps, to signals. A process can enable the delivery of any signal and establish a signal handler to deal with the consequences of the signal processwide.

The *Calling Standard for Alpha Systems* manual defines special structures and mechanisms that enable the processing of exceptional events on Tru64 UNIX systems in a more precise and organized way. Among the activities that the standard defines are the following:

- The manner in which exception handlers are established
- The way in which exceptions are raised
- How the exception system searches for and invokes a handler
- How a handler returns to the exception system
- The manner in which the exception system traverses the stack and maintains procedure context

The run-time exception dispatcher that supports the structured exception-handling capabilities of the Tru64 UNIX C compiler is an example of the type of frame-based exception handler described in the standard. (See Section 11.3 for a discussion of structured exception handling.)

The following sections briefly describe the Tru64 UNIX components that support the exception-handling mechanism defined in the *Calling Standard for Alpha Systems* manual.

11.1.1 C Compiler Syntax

Syntax provided by the Tru64 UNIX C compiler allows you to protect regions of code against user- or system-defined exception conditions. This mechanism, known as structured exception handling, allows you to define exception handlers and termination handlers and to indicate the regions of code that they protect.

The `c_excpt.h` header file defines the symbols and functions that user exception-processing code can use to obtain the current exception code and other information describing the exception.

11.1.2 libexc Library Routines

The exception support library, `/usr/ccs/lib/cmplrs/cc/libexc.a`, provides routines with the following capabilities:

- The ability to raise user-defined exceptions or convert UNIX signals to exceptions. These routines include:

  ```
  exc_raise_status_exception
  exc_raise_signal_exception
  exc_raise_exception
  exc_exception_dispatcher
  exc_dispatch_exception
  ```

  These exception-management routines also provide the mechanism to dispatch exceptions to the appropriate handlers. In the case of C language structured exception handling, described in Section 11.3, the C-specific handler invokes a routine containing user-supplied code to determine what action to take. The user-supplied code can either handle the exception or return for some other procedure activation to handle it.

- The ability to perform virtual and actual unwinding of levels of procedure activations from the stack and continuing execution in a handler or other user code. These routines include:

  ```
  unwind
  exc_virtual_unwind
  RtlVirtualUnwind
  ```

```
exc_resume
exc_longjmp
exc_continue
exc_unwind
RtlUnwindRfp
```

Some of the unwind routines also support invoking handlers as they unwind so that the language or user can clean up items at particular procedure activations.

- The ability to access procedure-specific information and map any address within a routine to the corresponding procedure information. This information includes enough data to cause an unwind or determine whether a routine handles an exception. These routines include:

```
exc_add_pc_range_table
exc_remove_pc_range_table
exc_lookup_function_table_address
exc_lookup_function_entry
find_rpd
exc_add_gp_range
exc_remove_gp_range
exc_lookup_gp
```

The C language structured exception handler calls routines in the last two categories to allow user code to fix up an exception and resume execution, and to locate and dispatch to a user-defined exception handler. Section 11.3 describes this process. For more information on any routine provided in `/usr/ccs/lib/cmplrs/cc/libexc.a`, see the routine's reference page.

11.1.3 Header Files That Support Exception Handling

Various header files define the structures that support the exception-handling system and the manipulation of procedure context. Table 11–1 describes these files.

Table 11–1: Header Files That Support Exception Handling

File	Description
`excpt.h`	Defines the exception code structure and defines a number of Tru64 UNIX exception codes; also defines the system exception and context records and associated flags and symbolic constants, the run-time procedure type, and prototypes for the functions provided in `libexc.a`. See `excpt`(4) for more information.

Table 11–1: Header Files That Support Exception Handling (cont.)

File	Description
c_excpt.h	Defines symbols used by C language structured exception handlers and termination handlers; also defines the exception information structure and functions that return the exception code, other exception information, and information concerning the state in which a termination handler is called. See c_excpt(4) for more information.
machine/fpu.h	Defines prototypes for the ieee_set_fp_control and ieee_get_fp_control routines, which enable the delivery of IEEE floating-point exceptions and retrieve information that records their occurrence; also defines structures and constants that support these routines. See ieee(3) for more information.
pdsc.h	Defines structures, such as the run-time procedure descriptor and code-range descriptor, that provide run-time contexts for the procedure types and flow-control mechanisms described in the *Calling Standard for Alpha Systems* manual. See pdsc(4) for more information.

11.2 Raising an Exception from a User Program

A user program typically raises an exception in either of two ways:

- A program can explicitly initiate an application-specific exception by calling the exc_raise_exception or exc_raise_status_exception function. These functions allow the calling procedure to specify information that describes the exception.
- A program can install a special signal handler, exc_raise_signal_exception, that converts a POSIX signal to an exception. The exc_raise_signal_exception function invokes the exception dispatcher to search the run-time stack for any exception handlers that have been established in the current or previous stack frames. In this case, the code reported to the handler has EXC_SIGNAL in its facility field and the signal value in its code field. (See excpt(4) and the excpt.h header file for a dissection of the code data structure.)

Note

The exact exception code for arithmetic and software-generated exceptions, defined in the signal.h header file, is passed to a signal handler in the *code* argument. The special signal handler exc_raise_signal_exception moves this code to ExceptionRecord.ExceptionInfo[0] before invoking the exception dispatcher.

The examples in Section 11.3 show how to explicitly raise an exception and convert a signal to an exception.

11.3 Writing a Structured Exception Handler

The structured exception-handling capabilities provided by the Tru64 UNIX C compiler allow you to deal with the possibility that a certain exception condition may occur in a certain code sequence. These capabilities are always enabled (the `cc` command's `-ms` option is not required). The syntax establishing a structured exception handler is as follows:

```
try {
  try-body

}
except ( exception-filter) {
  exception-handler

}
```

The *`try-body`* is a statement or block of statements that the exception handler protects. If an exception occurs while the try body is executing, the C-specific run-time handler evaluates the *`exception-filter`* to determine whether to transfer control to the associated *`exception-handler`*, continue searching for a handler in outer-level try body, or continue normal execution from the point at which the exception occurred.

The *`exception-filter`* is an expression associated with the exception handler that guards a given try body. It can be a simple expression or it can invoke a function that evaluates the exception. An exception filter must evaluate to one of the following integral values in order for the exception dispatcher to complete its servicing of the exception:

- < 0

  The exception dispatcher dismisses the exception and resumes the thread of execution that was originally disrupted by the exception. If the exception is noncontinuable, the dispatcher raises a `STATUS_NONCONTINUABLE_EXCEPTION` exception.

- 0

  The exception dispatcher continues to search for a handler, first in any `try...except` blocks in which the current handler might be nested and then in the `try...except` blocks defined in the procedure frame preceding the current frame on the run-time stack. If a filter chooses not to handle an exception, it typically returns this value.

- > 0

The exception dispatcher transfers control to the exception handler, and execution continues in the frame on the run-time stack in which the handler is found. This process, known as handling the exception, unwinds all procedure frames below the current frame and causes any termination handlers established within those frames to execute.

Two intrinsic functions are allowed within the exception filter to access information about the exception being filtered:

```
long                  exception_code ();
Exception_info_ptr    exception_info ();
```

The `exception_code` function returns the exception code. The `exception_info` function returns a pointer to an `EXCEPTION_POINTERS` structure. Using this pointer, you can access the machine state (for instance, the system exception and context records) at the time of the exception. See `excpt`(4) and `c_excpt`(4) for more information.

You can use the `exception_code` function within an exception filter or exception handler. However, you can use the `exception_info` function only within an exception filter. If you need to use the information returned by the `exception_info` function within the exception handler, invoke the function within the filter and store the information locally. If you need to refer to exception structures outside of the filter, you must copy them as well because their storage is valid only during the execution of the filter.

When an exception occurs, the exception dispatcher virtually unwinds the run-time stack until it reaches a frame for which a handler has been established. The dispatcher initially searches for an exception handler in the stack frame that was current when the exception occurred.

If the handler is not in this stack frame, the dispatcher virtually unwinds the stack (in its own context), leaving the current stack frame and any intervening stack frames intact until it reaches a frame that has established an exception handler. It then executes the exception filter associated with that handler.

During this phase of exception dispatching, the dispatcher has only virtually unwound the run-time stack; all call frames that may have existed on the stack at the time of the exception are still there. If it cannot find an exception handler or if all handlers reraise the exception, the exception dispatcher invokes the system last-chance handler. (See `exc_set_last_chance_handler`(3) for instructions on how to set up a last-chance handler.)

By treating the exception filter as if it were a Pascal-style nested procedure, exception-handling code evaluates the filter expression within the scope of the procedure that includes the `try...except` block. This allows the filter expression to access the local variables of the procedure containing the filter, even though the stack has not actually been unwound to the stack frame of the procedure that contains the filter.

Prior to executing an exception handler (for instance, if an exception filter returns EXCEPTION_EXECUTE_HANDLER), the exception dispatcher performs a real unwind of the run-time stack, executing any termination handlers established for `try...finally` blocks that terminated as a result of the transfer of control to the exception handler. Only then does the dispatcher call the exception handler.

The *`exception-handler`* is a compound statement that deals with the exception condition. It executes within the scope of the procedure that includes the `try...except` construct and can access its local variables. A handler can respond to an exception in several different ways, depending on the nature of the exception. For instance, it can log an error or correct the circumstances that led to the exception being raised.

Either an exception filter or exception handler can take steps to modify or augment the exception information it has obtained and ask the C language exception dispatcher to deliver the new information to exception code established in some outer try body or prior call frame. This activity is more straightforward from within the exception filter, which operates with the frames of the latest executing procedures — and the exception context — still intact on the run-time stack. The filter completes its processing by returning a 0 to the dispatcher to request the dispatcher to continue its search for the next handler.

For an exception handler to trigger a previously established handler, it must raise another exception, from its own context, that the previously established handler is equipped to handle.

Example 11–1 shows a simple exception handler established to handle a segmentation violation signal (SIGSEGV) that has been converted to an exception by the `exc_raise_signal_exception` signal handler.

Example 11–1: Handling a SIGSEGV Signal as a Structured Exception

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <excpt.h>

void main(void)
{
    struct sigaction    act, oldact;
    char                *x=0;

/*
 * Set up things so that SIGSEGV signals are delivered. Set
 * exc_raise_signal_exception as the SIGSEGV signal handler
 * in sigaction.
 */
```

Example 11–1: Handling a SIGSEGV Signal as a Structured Exception (cont.)

```
    act.sa_handler = exc_raise_signal_exception;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if (sigaction(SIGSEGV, &act, &oldact) < 0)
        perror("sigaction:");
/*
 * If a segmentation violation occurs within the following try
 * block, the run-time exception dispatcher calls the exception
 * filter associated with the except statement to determine
 * whether to call the exception handler to handle the SIGSEGV
 * signal exception.
 */
    try {
        *x=55;
    }
    /*
     * The exception filter tests the exception code against
     * SIGSEGV. If it tests true, the filter returns 1 to the
     * dispatcher, which then executes the handler; if it tests
     * false, the filter returns -1 to the dispatcher, which
     * continues its search for a handler in the previous run-time
     * stack frames. Eventually the last-chance handler executes.
     * Note: Normally the printf in the filter would be replaced
     * with a call to a routine that logged the unexpected signal.
     */
    except(exception_code() == EXC_VALUE(EXC_SIGNAL,SIGSEGV) ? 1 :
           (printf("unexpected signal exception code 0x%lx\n",
                   exception_code()), 0))
        {
            printf("segmentation violation reported: handler\n");
            exit(0);
        }
    printf("okay\n");
    exit(1);
}
```

The following is a sample run of this program:

```
% cc -std0 segfault_ex.c -lexc
% a.out
segmentation violation reported in handler
```

Example 11–2 is similar to Example 11–1 insofar as it also demonstrates a way of handling a signal exception, in this case, a SIGFPE. This example further shows how an IEEE floating-point exception, floating divide-by-zero, must be enabled by a call to `ieee_set_fp_control`(), and how the handler obtains more detailed information on the exception by reading the system exception record.

Example 11–2: Handling an IEEE Floating-Point SIGFPE as a Structured Exception

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <excpt.h>
#include <machine/fpu.h>
#include <errno.h>

int main(void)
{
    Exception_info_ptr  except_info;
    system_exrec_type   exception_record;
    long                code;
    struct sigaction    act, oldact;
    unsigned long       float_traps=IEEE_TRAP_ENABLE_DZE;
    double              temperature=75.2, divisor=0.0, quot, return_val;

/*
 * Set up things so that IEEE DZO traps are reported and that
 * SIGFPE signals are delivered.  Set exc_raise_signal_exception
 * as the SIGFPE signal handler.
 */
    act.sa_handler = exc_raise_signal_exception;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if (sigaction(SIGFPE, &act, &oldact) < 0)
        perror("sigaction:");
    ieee_set_fp_control(float_traps);

/*
 * If a floating divide-by-zero FPE occurs within the following
 * try block, the run-time exception dispatcher calls the
 * exception filter associated with the except statement to
 * determine whether the SIGFPE signal exception is to be
 * handled by the exception handler.
 */

    try {
        printf("quot = IEEE %.2f / %.2f\n",temperature,divisor);
        quot = temperature / divisor;
    }
/*
 * The exception filter saves the exception code and tests it
 * against SIGFPE. If it tests true, the filter obtains the
 * exception information, copies the exception record structure,
 * and returns 1 to the dispatcher which then executes the handler.
 * If the filter’s test of the code is false, the filter
 * returns 0 to the handler, which continues its search for a
 * handler in previous run-time frames. Eventually the last-chance
 * handler executes. Note: Normally the filter printf is replaced
 * with a call to a routine that logged the unexpected signal.
 */
    except((code=exception_code()) == EXC_VALUE(EXC_SIGNAL,SIGFPE) ?
           (except_info = exception_info(),
            exception_record = *(except_info->ExceptionRecord), 1) :
           (printf("unexpected signal exception code 0x%lx\n",
                   exception_code()), 0))
/*
 * The exception handler follows and prints out the signal code,
 * which has the following format:
 *
```

Example 11–2: Handling an IEEE Floating-Point SIGFPE as a Structured Exception (cont.)

```
 * 0x       8          0ffe              0003
 * |        |           |                 |
 * hex      SIGFPE     EXC_OSF facility  EXC_SIGNAL
 */
        { printf("Arithmetic error\n");
        printf("exception_code() returns 0x%lx\n", code);
        printf("EXC_VALUE macro in excpt.h generates 0x%lx\n",
               EXC_VALUE(EXC_SIGNAL, SIGFPE));
        printf("Signal code in the exception record is 0x%lx\n",
               exception_record.ExceptionCode);
/*
 * To find out what type of SIGFPE this is, look at the first
 * optional parameter in the exception record. Verify that it is
 * FPE_FLTDIV_TRAP).
 */
        printf("No. of parameters is %lu\n",
               exception_record.NumberParameters);
        printf("SIGFPE type is 0x%lx\n",
               exception_record.ExceptionInformation[0]);
/*
 * Set return value to IEEE_PLUS_INFINITY and return.
 */
        if (exception_record.ExceptionInformation[0] ==
            FPE_FLTDIV_TRAP)
            {
                *((long*)&return_val) = IEEE_PLUS_INFINITY;
                printf("Returning 0x%f to caller\n", return_val);
                return 0;
            }
/*
 * If this is a different kind of SIGFPE, return gracelessly.
 */
        else
            return -1;
        }
/*
 * We get here only if no exception occurred in the try block.
 */
    printf("okay: %f\n", quot);
    exit(1);
}
```

The following is a sample run of this program:

```
% cc -std0 sigfpe_ex.c -lexc
% a.out
quot  = IEEE 75.20 / 0.00
Arithmetic error
exception_code() returns 0x80ffe0003
The EXC_VALUE macro in excpt.h generates 0x80ffe0003
The signal code in the exception record is 0x80ffe0003
No. of parameters is 1
SIGFPE type is 0x4
Returning 0xINF to caller
```

A procedure (or group of interrelated procedures) can contain any number of `try...except` constructs, and can nest these constructs. If an exception occurs within the `try...except` block, the system invokes the exception handler associated with that block.

Example 11–3 demonstrates the behavior of multiple `try...except` blocks by defining two private exception codes and raising either of these two exceptions within the innermost try block.

Example 11–3: Multiple Structured Exception Handlers

```
#include <excpt.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define EXC_NOTWIDGET EXC_VALUE(EXC_C_USER, 1)
#define EXC_NOTDECWIDGET EXC_VALUE(EXC_C_USER, 2)

void getwidgetbyname(char *);

/*
 * main() sets up an exception handler to field the EXC_NOTWIDGET
 * exception and then calls getwidgetbyname().
 */
void main(int argc, char *argv[])
{
    char widget[20];
    long code;
    try {
        if (argc > 1)
            strcpy(widget, argv[1]);
        else
            {
                printf("Enter widget name: ");
                gets(widget);
            }
        getwidgetbyname(widget);
    }
    except((code=exception_code()) == EXC_NOTWIDGET)
        {
            printf("Exception 0x%lx: %s is not a widget\n",
                   code, widget);
            exit(0);
        }
}
/*
 * getwidgetbyname() sets up an exception handler to field the
 * EXC_NOTDECWIDGET exception. Depending upon the data it is
 * passed, its try body calls exc_raise_status_exception() to
```

Example 11–3: Multiple Structured Exception Handlers (cont.)

```
 * generate either of the user-defined exceptions.
 */
void
getwidgetbyname(char* widgetname)
{
    long  code;
    try {
        if (strcmp(widgetname, "foo") == 0)
            exc_raise_status_exception(EXC_NOTDECWIDGET);
        if (strcmp(widgetname, "bar") == 0)
            exc_raise_status_exception(EXC_NOTWIDGET);
    }
/*
 * The exception filter tests the exception code against
 * EXC_NOTDECWIDGET. If it tests true, the filter returns 1
 * to the dispatcher; if it tests false, the filter returns -1
 * to the dispatcher, which continues its search for a handler
 * in the previous run-time stack frames. When the generated
 * exception is EXC_NOTWIDGET, the dispatcher finds its handler
 * in main()'s frame.
 */
    except((code=exception_code()) == EXC_NOTDECWIDGET)
        {
            printf("Exception 0x%lx: %s is not "
                   "a Hewlett-Packard-supplied widget\n",
                   code, widgetname);
            exit(0);
        }
    printf("widget name okay\n");
}
```

The following is a sample run of this program:

```
% cc raise_ex.c -lexc
% a.out
Enter widget name: foo
Exception 0x20ffe009: foo is not a Hewlett-Packard-supplied widget
% a.out
Enter widget name: bar
Exception 0x10ffe009: bar is not a widget
```

11.4 Writing a Termination Handler

The `cc` compiler allows you to ensure that a specified block of termination code is executed whenever control is passed from a guarded body of code.

The termination code is executed regardless of how the flow of control leaves the guarded code. For example, a termination handler can guarantee that cleanup tasks are performed even if an exception or some other error occurs while the guarded body of code is executing.

The syntax for a termination handler is as follows:

```
try {
  try-body

}
finally {
    termination-handler

}
```

The `try-body` is the code, expressed as a compound statement, that the termination handler protects. The try body can be a block of statements or a set of nested blocks. It can include the following statement, which causes an immediate exit from the block and execution of its termination handler:

leave;

Note

> The `longjmp()` routine for Tru64 UNIX does not use an unwind operation. Therefore, in the presence of frame-based exception handling, do not use a `longjmp()` from a *`try-body`* or *`termination-handler`*. Rather, use an `exc_longjmp()`, which is implemented through an unwind operation.

The *`termination-handler`* is a compound statement that executes when the flow of control leaves the guarded try body, regardless of whether the try body terminated normally or abnormally. The guarded body is considered to have terminated normally when the last statement in the block is executed (that is, when the body's closing "}" is reached). Using the `leave` statement also causes a normal termination. The guarded body terminates abnormally when the flow of control leaves it by any other means, for example, due to an exception or due to a control statement such as `return`, `goto`, `break`, or `continue`.

A termination handler can call the following intrinsic function to determine whether the guarded body terminated normally or abnormally:

int abnormal_termination ();

The `abnormal_termination` function returns 0 if the try body completed sequentially; otherwise, it returns 1.

The termination handler itself may terminate either sequentially or by a transfer of control out of the handler. If it terminates sequentially (by reaching the closing “}”), subsequent control flow depends on how the try body terminated:

- If the try body terminated normally, execution continues with the statement following the complete `try...finally` block.
- If the try body terminated abnormally with an explicit jump out of the body, the jump is completed. However, if the jump exits the body of one or more containing `try...finally` statements, their termination handlers are invoked before control is finally transferred to the target of the jump.
- If the try body terminated abnormally due to an unwind, a jump to an exception handler, or an `exc_longjmp` call, control is returned to the C run-time exception handler, which will continue invoking termination handlers as required before jumping to the target of the unwind.

Like exception filters, termination handlers are treated as Pascal-style nested procedures and are executed without the removal of frames from the run-time stack. A termination handler can thus access the local variables of the procedure in which it is declared.

Note that there is a performance cost in the servicing of abnormal terminations, inasmuch as abnormal terminations (and exceptions) are considered to be outside the normal flow of control for most programs. Keep in mind that explicit jumps out of a try body are considered abnormal termination. Normal termination is the simple case and costs less at run time.

In some instances, you can avoid this cost by replacing a jump out of a try body with a `leave` statement (which transfers control to the end of the innermost try body) and testing a status variable after completion of the entire `try...finally` block.

A termination handler itself may terminate nonsequentially (for instance, to abort an unwind) by means of a transfer of control (for instance, a `goto`, `break`, `continue`, `return`, `exc_longjmp`, or the occurrence of an exception). If this transfer of control exits another `try...finally` block, its termination handler will execute.

Example 11–4 shows the order in which termination handlers and exception handlers execute when an exception causes the termination of the innermost try body.

Example 11–4: Abnormal Termination of a Try Block by an Exception

```
#include <stdio.h>
#include <signal.h>
#include <excpt.h>
#include <errno.h>

#define EXC_FOO EXC_VALUE(EXC_C_USER, 1)

signed int
foo_except_filter(void)
{
    printf("2. The exception causes the exception filter "
           "to be evaluated.\n");
    return 1;
}

void main (void)
{
    try {
        try {
            printf("1. The main body executes.\n");
            exc_raise_status_exception(EXC_FOO);
        }
        finally {
            printf("3. The termination handler executes "
                   "because control will leave the "
                   "try...finally block to \n");
        }
    }
    except(foo_except_filter()) {
        printf("4. execute the exception handler.\n");
    }
}
```

The following is a sample run of this program:

```
% cc segfault_ex2.c -lexc
% a.out
1. The main body executes.
2. The exception causes the exception filter to be evaluated.
3. The termination handler executes because control will leave the
   try...finally block to
4. execute the exception handler.
```

12

Developing Thread-Safe Libraries

To support the development of multithreaded applications, the Tru64 UNIX operating system provides the POSIX Threads Library, the Compaq Multithreading Run-Time Library. The POSIX Threads Library interface implements IEEE Standard 1003.1c-1995 threads (also referred to as POSIX 1003.1c threads), with several extensions.

In addition to an actual threading interface, the operating system also provides Thread-Independent Services (TIS). The TIS routines are an aid to creating efficient thread-safe libraries that do not create their own threads. (See Section 12.4.1 for information about TIS routines.)

This chapter addresses the following topics:

- Overview of multithread support in Tru64 UNIX (Section 12.1)
- Run-time library changes for POSIX conformance (Section 12.2)
- Characteristics of thread-safe and thread-reentrant routines (Section 12.3)
- How to write thread-safe code (Section 12.4)
- How to build multithreaded applications (Section 12.5)

12.1 Overview of Thread Support

A thread is a single, sequential flow of control within a program. Multiple threads execute concurrently and share most resources of the owning process, including the address space. By default, a process initially has one thread.

The purposes for which multiple threads are useful include:

- Improving the performance of applications running on multiprocessor systems
- Implementing certain programming models (for example, the client/server model)
- Encapsulating and isolating the handling of slow devices

You can also use multiple threads as an alternative approach to managing certain events. For example, you can use one thread per file descriptor in a process that otherwise might use the `select()` or `poll()` system calls to efficiently manage concurrent I/O operations on multiple file descriptors.

The components of the multithreaded development environment for the Tru64 UNIX system include the following:

- Compiler support — Compile using the `−pthread` option on the `cc` or `c89` command.
- Threads package — The `libpthread.so` library provides interfaces for threads control.
- Thread-safe support libraries — These libraries include `libaio`, `libcfg`, `liblmf`, `libm`, `libmsfs`, `libpruplist`, `libpthread`, `librt`, and `libsys5`.
- The `ladebug` debugger
- The `prof` and `gprof` profilers — Compile with the `−p` and `−pthread` options for `prof` and with `−pg` and `−pthread` for `gprof` to use the `libprof1_r.a` profiling library.
- The `atom`-based `pixie`, `hiprof`, and `third` tools

For information on profiling multithreaded applications, see Section 8.8.

To analyze a multithreaded application for potential logic and performance problems, you can use Visual Threads, which is available on the Associated Products Volume 1 CD-ROM. Visual Threads can be used on POSIX Threads Library applications and on Java applications.

12.2 Run-Time Library Changes for POSIX Conformance

For releases of the DEC OSF/1 system (that is, for releases prior to DIGITAL UNIX Version 4.0), a large number of separate reentrant routines (*`_r` routines) were provided to solve the problem of static data in the C run-time library (the first two problems listed in Section 12.3.1). For releases of the Tru64 UNIX system, the problem of static data in the nonreentrant versions of the routines is fixed by replacing the static data with thread-specific data. Except for a few routines specified by POSIX 1003.1c, the alternate routines are not needed on Tru64 UNIX systems and are retained only for binary compatibility.

The following functions are the only alternate thread-safe routines that are specified by POSIX 1003.1c and need to be used when writing thread-safe code:

```
asctime_r*          ctime_r*            getgrgid_r*

getgrnam_r*         getpwnam_r*         getpwuid_r*
```

gmtime_r*
localtime_r*
rand_r*
readdir_r*
strtok_r

Starting with DIGITAL UNIX Version 4.0, the interfaces flagged with an asterisk (*) in the preceding list have new definitions that conform to POSIX 1003.1c. The old versions of these routines can be obtained by defining the preprocessor symbol `_POSIX_C_SOURCE` with the value `199309L` (which denotes POSIX 1003.1b conformance — however, doing this will disable POSIX 1003.1c threads). The new versions of the routines are the default when compiling code under DIGITAL UNIX Version 4.0 or higher, but you must be certain to include the header files specified on the reference pages for the various routines.

For more information on programming with threads, see the *Guide to the POSIX Threads Library* and `cc`(1), `monitor`(3), `prof`(1), and `gprof`(1).

12.3 Characteristics of Thread-Safe and Reentrant Routines

Routines within a library can be thread-safe or not. A thread-safe routine is one that can be called concurrently from multiple threads without undesirable interactions between threads. A routine can be thread-safe for either of the following reasons:

- It is inherently reentrant.
- It uses thread-specific data or mutex locks. (A mutex is a synchronization object that is used to allow multiple threads to serialize their access to shared data.)

Reentrant routines do not share any state across concurrent invocations from multiple threads. A reentrant routine is the ideal thread-safe routine, but not all routines can be made reentrant.

Prior to DIGITAL UNIX Version 4.0, many of the C run-time library (`libc`) routines were not thread-safe, and alternate versions of these routines were provided in `libc_r`. Starting with DIGITAL UNIX Version 4.0, all of the alternate versions formerly found in `libc_r` were merged into `libc`. If a thread-safe routine and its corresponding nonthread-safe routine had the same name, the nonthread-safe version was replaced. The thread-safe versions are modified to use TIS routines (see Section 12.4.1); this enables them to work in both single-threaded and multithreaded environments — without extensive overhead in the single-threaded case.

12.3.1 Examples of Nonthread-Safe Coding Practices

Some common practices that can prevent code from being thread-safe can be found by examining why some of the libc functions were not thread-safe prior to DIGITAL UNIX Version 4.0:

- **Returning a pointer to a single, statically allocated buffer**

  The ctime(3) interface provides an example of this problem:

  ```
  char *ctime(const time_t *timer);
  ```

  This function takes no output arguments and returns a pointer to a statically allocated buffer containing a string that is the ASCII representation of the time specified in the single parameter to the function. Because a single, statically allocated buffer is used for this purpose, any other thread that calls this function will overwrite the string returned to the previously calling thread.

  To make the ctime() function thread-safe, the POSIX 1003.1c standard has defined an alternate version, ctime_r(), which accepts an additional output argument. The argument is a user-supplied buffer that is allocated by the caller. The ctime_r() function writes the ASCII time string into the buffer:

  ```
  char *ctime_r(const time_t *timer, char *buf);
  ```

  Users of this function must ensure that the storage occupied by the buf argument is not used by another thread.

- **Maintaining internal state**

  The rand() function provides an example of this problem:

  ```
  void srand(unsigned int seed);
  int rand(void);
  ```

  This function is a simple pseudorandom number generator. For any given starting seed value that is set with the srand() function, it generates an identical sequence of pseudorandom numbers. To do this, it maintains a state value that is updated on each call. If another thread is calling this function, the sequence of numbers returned within any one thread for a given starting seed is nondeterministic. This may be undesirable.

  To avoid this problem, a second interface, rand_r(), is specified in POSIX 1003.1c. This function accepts an additional argument that is a pointer to a user-supplied integer used by rand_r() to hold the state of the random number generator:

  ```
  int rand_r(unsigned int *seed);
  ```

  The users of this function must ensure that the seed argument is not used by another thread. Using thread-specific data is one way of doing this (see Section 12.4.1.2).

- **Operating on read/write data items shared between threads**

  The problem of sharing read/write data can be solved by using mutexes. In this case, the routine is not considered reentrant, but it is still thread-safe. Like thread-specific data, mutex locking is transparent to the user of the routine.

  Mutexes are used in several libc routines, most notably the stdio routines, for example, printf(). Mutex locking in the stdio routines is done by stream to prevent concurrent operations on a stream from colliding, as in the case of two processes trying to fill a stream buffer at the same time. Mutex locking is also done on certain internal data tables in the C run-time library during operations such as fopen() and fclose(). Because the alternate versions of these routines do not require an application program interface (API) change, they have the same name as the original versions.

  See Section 12.4.3 for an example of how to use mutexes.

12.4 Writing Thread-Safe Code

When writing code that can be used by both single-threaded and multithreaded applications, it is necessary to code in a thread-safe manner. The following coding practices must be observed:

- Static read/write data should be either eliminated, converted to thread-specific data, or protected by mutexes. In the C language, to reduce the potential for misuse of the data, it is good practice to declare static read-only data with the const type modifier.
- Global read/write data should be eliminated or protected by mutex locks.
- Per-process system resources, such as file descriptors, should be used with care because they are accessible by all threads.
- References to the global errno cell should be replaced with calls to geterrno() and seterrno(). This replacement is not necessary if the source file includes errno.h and one of the following conditions is true:
  - The file is compiled with the -pthread option (cc or c89 command).
  - The pthread.h file is included at the top of the source file.
  - The _REENTRANT preprocessor symbol is explicitly set before the include of errno.h.
- Dependencies on any other nonthread-safe libraries or object files must be avoided.

12.4.1 Using TIS for Thread-Specific Data

The following sections explain how to use Thread Independent Services (TIS) for thread-specific data.

12.4.1.1 Overview of TIS

Thread Independent Services (TIS) is a package of routines provided by the C run-time library that can be used to write efficient code for both single-threaded and multithreaded applications. TIS routines can be used for handling mutexes, handling thread-specific data, and a variety of other purposes.

When used by a single-threaded application, these routines use simplified semantics to perform thread-safe operations for the single-threaded case. When POSIX Threads Library is present, the bodies of the routines are replaced with more complicated algorithms to optimize their behavior for the multithreaded case.

TIS is used within `libc` itself to allow a single version of the C run-time library to service both single-threaded and multithreaded applications. See the *Guide to the POSIX Threads Library* and `tis`(3) for information on how to use this facility.

12.4.1.2 Using Thread-Specific Data

Example 12–1 shows how to use thread-specific data in a function that can be used by both single-threaded and multithreaded applications. For clarity, most error checking has been left out of the example.

Example 12–1: Threads Programming Example

```
#include <stdlib.h>
#include <string.h>
#include <tis.h>

static pthread_key_t key;

void __init_dirname()
{
 tis_key_create(&key, free);
}

void __fini_dirname()
{
 tis_key_delete(key);
}

char *dirname(char *path)
```

Example 12–1: Threads Programming Example (cont.)

```
{
 char *dir, *lastslash;
/*
 * Assume key was set and get thread-specific variable.
 */
 dir = tis_getspecific(key);
 if(!dir) { /* First time this thread got here. */
  dir = malloc(PATH_MAX);
  tis_setspecific(key, dir);
 }

/*
 * Copy dirname component of path into buffer and return.
 */
 lastslash = strrchr(path, ’/’);
 if(lastslash) {
  memcpy(dir, path, lastslash-path);
  dir[lastslash-dir+1] = ’\0’;
 } else
  strcpy(dir, path);
 return dir;
}
```

The following TIS routines are used in the preceding example:

`tis_key_create`

Generates a unique data key.

`tis_key_delete`

Deletes a data key.

`tis_getspecific`

Obtains the data associated with the specified key.

`tis_setspecific`

Sets the data value associated with the specified key.

The `__init_` and `__fini_` routines are used in the example to initialize and destroy the thread-specific data key. This operation is done only once, and these routines provide a convenient way to ensure that this is the case, even if the library is loaded with `dlopen()`. See `ld`(1) for an explanation of how to use the `__init_` and `__fini_` routines.

Thread-specific data keys are provided by POSIX Threads Library at run time and are a limited resource. If your library must use a large number of data keys, code the library to create just one data key and store all of the separate data items as a structure or an array of pointers pointed to by that key.

12.4.2 Using Thread Local Storage

Thread Local Storage (TLS) support is always enabled in the C compiler (the `cc` command's `-ms` option is not required). In C++, TLS is recognized only with the `-ms` option, and it is otherwise treated as an error.

TLS is a name for data that has static extent (that is, not on the stack) for the lifetime of a thread in a multithreaded process, and whose allocation is specific to each thread.

In standard multithreaded programs, static-extent data is shared among all threads of a given process, whereas thread local storage is allocated on a per-thread basis such that each thread has its own copy of the data that can be modified by that thread without affecting the value seen by the other threads in the process. For a complete discussion of threads, see the *Guide to the POSIX Threads Library*.

The essential functionality of thread local storage is and has been provided by explicit application program interfaces (APIs) such as POSIX (POSIX Threads Library) `pthread_key_create()`, `pthread_setspecific()`, `pthread_getspecific()`, and `pthread_key_delete()`.

Although these APIs are portable to POSIX-conforming platforms, using them can be cumbersome and error-prone. Also, significant engineering work is typically required to take existing single-threaded code and make it thread-safe by replacing all of the appropriate `static` and `extern` variable declarations and their uses by calls to these thread-local APIs. Furthermore, for Windows-32 platforms there is a somewhat different set of APIs (`TlsAlloc()`, `TlsGetValue()`, `TlsSetValue()`, and `TlsFree()`), which have the same kinds of usability problems as the POSIX APIs.

By contrast, the TLS language feature is much simpler to use than any of the APIs, and it is especially convenient to use when converting single-threaded code to be multithreaded. This is because the change to make a `static` or `extern` variable have a thread-specific value only involves adding a storage-class qualifier to its declaration. The compiler, linker, program loader, and debugger effectively implement the complexity of the API calls automatically for variables declared with this qualifier. Unlike coding to the APIs, it is not necessary to find and modify all uses of the variables, or to add explicit allocation and deallocation code. While the language feature is not generally portable under any formal programming standard, it is portable between Tru64 UNIX and Windows-32 platforms.

12.4.2.1 The __thread Attribute

The C and C++ compilers for Tru64 UNIX include the extended storage-class attribute, `__thread`.

The `__thread` attribute must be used with the `__declspec` keyword to declare a thread variable. For example, the following code declares an integer thread local variable and initializes it with a value:

```
__declspec( __thread ) int tls_i = 1;
```

12.4.2.2 Guidelines and Restrictions

You must observe these guidelines and restrictions when declaring thread local objects and variables:

- You can apply the `__thread` storage-class attribute only to data declarations and definitions. It cannot be used on function declarations or definitions. For example, the following code generates a compiler error:

  ```
  #define Thread __declspec( __thread )
  Thread void func();           // Error
  ```

- You can specify the `__thread` attribute only on data items with static storage duration. This includes global data objects (both `static` and `extern`), local `static` objects, and `static` data members of C++ classes. You cannot declare automatic or register data objects with the `__thread` attribute. For example, the following code generates compiler errors:

  ```
  #define Thread  __declspec( __thread )
  void func1()
  {
     Thread int tls_i;            // Error
  }

  int func2( Thread int tls_i )    // Error
  {
     return tls_i;
  }
  ```

- You must use the `__thread` attribute for the declaration and the definition of a thread-local object, whether the declaration and definition occur in the same file or separate files. For example, the following code generates an error:

  ```
  #define Thread          __declspec( __thread )
  extern int tls_i;      // This generates an error, because the
  int Thread tls_i;      // declaration and the definition differ.
  ```

- You cannot use the `__thread` attribute as a type modifier. For example, the following code generates a compile-time error:

  ```
  char __declspec( __thread ) *ch;            // Error
  ```

- The address of a thread-local object is not considered a link-time constant, and any expression involving such an address is not considered a constant expression. In standard C, the effect of this is to forbid the use of the address of a thread-local variable as an initializer for an object that has static or thread-local extent. For example, the following code is flagged as an error by the C compiler if it appears at file scope:

  ```
  #define Thread  __declspec( __thread )
  Thread int tls_i;
  int *p = &tls_i;            // ERROR
  ```

- Standard C permits initialization of an object or variable with an expression involving a reference to itself, but only for objects of nonstatic extent. Although C++ normally permits such dynamic initialization of an object with an expression involving a reference to itself, this type of initialization is not permitted with thread local objects. For example:

  ```
  #define Thread  __declspec( __thread )
  Thread int tls_i = tls_i;            // C and C++ error
  int j = j;                           // Okay in C++; C error
  Thread int tls_i = sizeof( tls_i )   // Okay in C and C++
  ```

  Note that a `sizeof` expression that includes the object being initialized does not constitute a reference to itself and is allowed in C and C++.

12.4.3 Using Mutex Locks to Share Data Between Threads

In some cases, using thread-specific data is not the correct way to convert static data into thread-safe code. For example, thread-specific data should not be used when a data object is meant to be shareable between threads (as in `stdio` streams within `libc`). Manipulating per-process resources is another case in which thread-specific data is inadequate. The following example shows how to manipulate per-process resources in a thread-safe fashion:

```
#include <pthread.h>
#include <tis.h>

/*
 * NOTE: The putenv() function would have to set and clear the
 * same mutex lock before it accessed the environment.
 */

extern char **environ;
static pthread_mutex_t environ_mutex = PTHREAD_MUTEX_INITIALIZER;

char *getenv(const char *name)
{
        char **s, *value;
       int len;
        tis_mutex_lock(&environ_mutex);
        len = strlen(name);
```

```
        for(s=environ; value=*s; s++)
                if(strncmp(name, value, len) == 0 &&
                   value[len] == '=') {
                        tis_mutex_unlock(&environ_mutex);
                        return &(value[len+1]);
                }
        tis_mutex_unlock(&environ_mutex);
        return (char *) 0L;
}
```

In the preceding example, note how the lock is set once (`tis_mutex_lock`) before accessing the environment and is unlocked exactly once (`tis_mutex_unlock`) before returning. In the multithreaded case, any other thread attempting to access the environment while the first thread holds the lock is blocked until the first thread performs the unlock operation. In the single-threaded case, no contention occurs unless an error exists in the coding of the locking and unlocking sequences.

If it is necessary for the lock state to remain valid across a `fork()` system call in multithreaded applications, it may be useful to create and register `pthread_atfork()` handler functions to set the lock prior to any `fork()` call, and to unlock it in both the child and parent after the `fork()` call. This guarantees that a `fork()` operation is not done by one thread while another thread holds the lock. If the lock was held by another thread, it would end up permanently locked in the child because the `fork()` operation produces a child with only one thread. In the case of an independent library, the call to `pthread_atfork()` can be done in an `__init_` routine in the library. Unlike most Pthread routines, the `pthread_atfork` routine is available in `libc` and may be used by both single-threaded and multithreaded applications.

12.5 Building Multithreaded Applications

The compilation and linking of multithreaded applications differs from that of single-threaded applications in a few minor but important ways. The following sections describe these differences.

12.5.1 Compiling Multithreaded C Applications

Depending on whether an application is single-threaded or multithreaded, many system header files provide different sets of definitions when they are included in the compilation of an application. Whether the compiler generates single-threaded or thread-safe behavior is determined by whether the `_REENTRANT` preprocessor symbol is defined. When you specify the `–pthread` option on the `cc` or `c89` command, the `_REENTRANT` symbol is automatically defined; it is also defined if the `pthread.h` header file is included. This header file must be the first file included in any application that uses the Pthread library, libpthread.so.

The −pthread option has no other effect on the compilation of C programs. The reentrancy of the code generated by the C compiler is determined only by the proper use of reentrant coding practices by the programmer and by the use of thread-safe support routines or functions — not by the use of any special options.

12.5.2 Linking Multithreaded C Applications

To link a multithreaded C application, use the cc or c89 command with the −pthread option. When linking, the −pthread option has the effect of modifying the library search path in the following ways:

- The Pthread library is included into the link.
- The exceptions library is included into the link.
- For each library mentioned in a −l option, an attempt is made to locate and presearch a library of corresponding thread-safe routines whose name includes the suffix _r.

The −pthread option does not modify the behavior of the linker in any other way. The reentrancy of the linked code is determined by using proper programming practices in the orginal code, and by compiling and linking with the proper header files and libraries, respectively.

12.5.3 Building Multithreaded Applications in Other Languages

Not all compilers necessarily generate reentrant code; the definition of the language itself can make this difficult. It is also necessary for any run-time libraries linked with the application to be thread-safe. For details on such matters, consult the manual for the compiler you are using and the documentation for the run-time libraries.

13

OpenMP Parallel Processing

The Comapq C compiler supports the development of shared memory parallel applications through its conformance to the OpenMP C Application Program Interface. This API defines a collection of compiler directives, library functions, and environment variables that instruct the compiler, linker, and run-time environment to perform portions of the application in parallel.

The OpenMP directives let you write code that can run concurrently on multiple processors without altering the structure of the source code from that of ordinary ANSI C serial code. Correct use of these directives can greatly improve the elapsed-time performance of user code by allowing that code to execute simultaneously on different processors of a multiprocessor machine. Compiling the same source code, but ignoring the parallel directives, produces a serial C program that performs the same function as the OpenMP compilation.

The *OpenMP C and C++ Application Programming Interface* specification is available on the Internet from `http://www.openmp.org/specs/`.

This chapter addresses the following topics:

- Compilation options (Section 13.1)
- Environment variables (Section 13.2)
- Run-time performance tuning (Section 13.3)
- Common programming problems (Section 13.4)
- Implementation-specific behavior (Section 13.5)
- Debugging (Section 13.6)

13.1 Compilation Options

The following options on the `cc` command line support parallel processing:

`-mp`

Causes the compiler to recognize both OpenMP manual decomposition pragmas and old-style manual decomposition directives. Forces `libots3` to be included in the link. (Old-style manual decomposition directives are described in Appendix D.)

`-omp`

Causes the compiler to recognize only OpenMP manual decomposition pragmas and to ignore old-style manual decomposition directives. (The `-mp` and `-omp` switches are the same except for their treatment of old-style manual decomposition directives; `-mp` recognizes the old-style directives and `-omp` does not.)

`-granularity` *`size`*

Controls the size of shared data in memory that can be safely accessed from different threads. Valid values for *`size`* are: `byte`, `longword`, and `quadword`:

`byte`

> Requests that all data of one byte or greater can be accessed from different threads sharing data in memory. This option will slow run-time performance.

`longword`

> Ensures that naturally aligned data of four bytes or greater can be accessed safely from different threads sharing access to that data in memory. Accessing data items of three bytes or less and unaligned data may result in data items written from multiple threads being inconsistently updated.

`quadword`

> Ensures that naturally aligned data of eight bytes can be accessed safely from different threads sharing data in memory. Accessing data items of seven bytes or less and unaligned data may result in data items written from multiple threads being inconsistently updated. This is the default.

`-check_omp`

Enables run-time checking of certain OpenMP constructs. This includes run-time detection of invalid nesting and other invalid OpenMP cases. When invalid nesting is discovered at run time and this switch is set, the executable will fail with a Trace/BPT trap. If this switch is not set and invalid nesting is discovered, the behavior is indeterminate (for example, an executable may hang).

The compiler detects the following invalid nesting conditions:

- Entering a `for`, `single`, or `sections` directive if already in a work-sharing construct, `critical` section, or a `master`
- Executing a `barrier` directive if already in a work-sharing sharing construct, a `critical` section, or a `master`
- Executing a `master` directive if already in a work-sharing construct
- Executing an `ordered` directive if already in a `critical` section
- Executing an `ordered` directive unless already in an `ordered for`

The default is disabled run-time checking.

13.2 Environment Variables

In addition to the environment variables outlined in the OpenMP specification, the following environment variables are recognized by the compiler and the run-time system:

`MP_THREAD_COUNT` Specifies how many threads are to be created by the run-time system. The default is the number of processors available to your process. The `OMP_NUM_THREADS` environment variable takes precedence over this variable.

`MP_STACK_SIZE` Specifies how many bytes of stack space are to be allocated by the run-time system for each thread. If you specify zero, the run-time system uses the default, which is very small. Therefore, if a program declares any large arrays to be PRIVATE, specify a value large enough to allocate them. If you do not use this environment variable, the run-time system allocates 5 MB.

`MP_SPIN_COUNT` Specifies how many times the run-time system can spin while waiting for a condition to become true.

The default is 16,000,000, which is approximately one second of CPU time.

`MP_YIELD_COUNT`
Specifies how many times the run-time system can alternate between calling `sched_yield` and testing the condition before going to sleep by waiting for a thread condition variable. The default is 10.

13.3 Run-Time Performance Tuning

The OpenMP specification provides a variety of methods for distributing work to the available threads within a parallel for construct. The following sections describe these methods.

13.3.1 Schedule Type and Chunksize Settings

The choice of settings for the schedule type and the chunksize can affect the ultimate performance of the resulting parallelized application, either positively or negatively. Choosing inappropriate settings for the schedule type and the chunksize can degrade the performance of parallelized application to the point where it performs as bad or worse than it would if it was serialized.

The general guidelines are as follows:

- Smaller chunksize values generally perform faster than larger. The values for the chunksize should be less than or equal to the values derived by dividing the number of iterations by the number of available threads.
- The behavior of the `dynamic` and `guided` schedule types make them better suited for target machines with a variety of workloads, other than the parallelized application. These types assign iterations to threads as they become available; if a processor (or processors) becomes tied up with other applications, the available threads will pick up the next iterations.
- Although the `runtime` schedule type does facilitate tuning of the schedule type at run time, it results in a minor performance penalty in run-time overhead.
- An effective means of determining appropriate settings for schedule and chunksize can be to set the schedule to `runtime` and experiment with various schedule and chunksize pairs through the `OMP_SCHEDULE` environment variable. After the exercise, explicitly set the schedule and chunksize to the values that yielded the best performance.

The schedule and chunksize settings are only two of the many factors that can affect the performance of your application. Some of the other areas that can affect performance include:

- Availability of system resources: CPUs on the target machine spending time processing other applications are not available to the parallelized application.
- Structure of parallelized code: Threads of a parallelized region that perform disproportionate amounts of work.
- Use of implicit and explicit barriers: Parallelized regions that force synchronization of all threads at these explicit or implicit points may cause the application to suspend while waiting for a thread (or threads).
- Use of `critical` sections versus `atomic` statements: Using `critical` sections incurs more overhead than `atomic`. For more information on schedule types and chunksize settings, see the *OpenMP C and C++ Application Programming Interface* specification.

13.3.2 Additional Controls

When one of the threads needs to wait for an event caused by some other thread, a three-level process begins:

1. The thread spins for a number of iterations waiting for the event to occur.
2. It yields the processor to other threads a number of times, checking for the event to occur.
3. It posts a request to be awakened and goes to sleep.

When another thread causes the event to occur, it will awaken the sleeping thread.

You may get better performance by tuning the threaded environment with the `MP_SPIN_COUNT` and `MP_YIELD_COUNT` environment variables or by using the `mpc_destroy` routine:

- `MP_SPIN_COUNT` — If your application is running standalone, the default settings will give good performance. However, if your application needs to share the processors with other applications, it is probably appropriate to reduce `MP_SPIN_COUNT`. This will make the threads waste less time spinning and give up the processor sooner; the cost is the extra time to put a thread to sleep and re-awaken it. In such a shared environment, an `MP_SPIN_COUNT` of about 1000 might be a good choice.
- `mpc_destroy` — If you need to perform operations that are awkward when extra threads are present (for example, `fork`), the `mpc_destroy` routine can be useful. It destroys any worker threads created to run parallel regions. Normally, you would only call it when you are not inside a parallel region. (The `mpc_destroy` routine is defined in the `libots3` library.)

13.4 Common Programming Problems

The following sections describe some errors that commonly occur in parallelized programs.

13.4.1 Scoping

The OpenMP parallel construct applies to the structured block that immediately follows it. When more than one statement is to be performed in parallel, make sure that the structured block is contained within curly braces. For example:

```
#pragma omp parallel
{
   pstatement one
   pstatement two
}
```

The preceding structured block is quite different from the following, where the OpenMP parallel construct applies to only the first statement:

```
#pragma omp parallel
   pstatement one
   pstatement two
```

The use of curly braces to explicitly define the scope of the subsequent block (or blocks) is strongly encouraged.

13.4.2 Deadlock

As with any multithreaded application, programmers must use care to prevent run-time deadlock conditions. With the implicit barriers at the end of many OpenMP constructs, an application will result in a deadlock if all threads do not actively participate in the construct. These types of conditions may be more prevalent when implementing parallelism in dynamic extents of the application. For example:

```
worker ()
 {
 #pragma omp barrier
 }

 main ()
 {
 #pragma omp parallel sections
   {
   #pragma omp section
     worker();
   }
 }
```

The preceding example results in deadlock (with more than one thread active) because not all threads visit the `worker` routine and the barrier waits for all threads. The `-check_omp` option (see Section 13.1) aids in detecting such conditions.

For more information, see the *OpenMP C and C++ Application Programming Interface* specification for a description of valid and invalid directive nesting.

13.4.3 Threadprivate Storage

The `threadprivate` directive identifies variables that have file scope but are private to each thread. The values for these variables are maintained if the number of threads remains constant. If you explicitly increase or decrease the number of threads within a program, the impact on the values for `threadprivate` variables is not defined.

13.4.4 Using Locks

Using the lock control routines (see the *OpenMP C and C++ Application Programming Interface* specification) requires that they be called in a specific sequence:

1. The lock to be associated with the lock variable must first be initialized.
2. The associated lock is made available to the executing thread.
3. The executing thread is released from lock ownership.
4. When finished, the lock must always be disassociated from the lock variable.

Attempting to use the locks outside the above sequence may cause unexpected behavior, including deadlock conditions.

13.5 Implementation-Specific Behavior

The OpenMP specification identifies several features and default values as implementation-specific. This section lists those instances and the implementation chosen by Compaq C.

Support for nested parallel regions

Whenever a nested parallel region is encountered, a team consisting of one thread is created to execute that region.

Default value for `OMP_SCHEDULE`

The default value is `dynamic,1`. If an application uses the run-time schedule but `OMP_SCHEDULE` is not defined, then this value is used.

Default value for `OMP_NUM_THREADS`

The default value is equal to the number of processors on the machine.

Default value for `OMP_DYNAMIC`

The default value is 0. This implementation does not support dynamic adjustments to the thread count.

Attempts to use omp_set_dynamic to a nonzero value have no effect on the run-time environment.

Default schedule

When a for or parallel for loop does not contain a schedule clause, a dynamic schedule type is used with the chunksize set to 1.

Flush directive

The flush directive, when encountered, will flush all variables, even if one or more variables are specified in the directive.

13.6 Debugging

The following sections provide tips and hints on how to diagnose the behavior of and debug applications that use the OpenMP application programming interface (API).

13.6.1 Background Information Needed for Debugging

The -mp or -omp options cause the compiler to recognize OpenMP directives and to transform specified portions of code into parallel regions. The compiler implements a parallel region by taking the code in the region and putting it into a separate, compiler-created routine. This process is called **outlining**, because it is the inverse of inlining a routine into source code at the point where the routine is called.

Note

Understanding how the parallel regions are outlined is necessary to effectively use the debugger and other application-analysis tools.

In place of the parallel region, the compiler inserts a call to a run-time library routine. The run-time library routine creates the slave threads in the team (if they were not already created), starts all threads in the team, and causes them to call the outlined routine. As threads return from the outlined routine, they return to the run-time library, which waits for all threads to finish before the master thread returns to the calling routine. While the master thread continues nonparallel execution, the slave threads wait, or spin, until either a new parallel region is encountered or until the environment-variable controlled wait time (MP_SPIN_COUNT) is reached. If the wait time expires, the slave threads are put to sleep until the next parallel region is encountered.

The following source code contains a parallel region in which the variable *id* is private to each thread. The code preceding the parallel region explicitly sets the number of threads used in the parallel region to 2. The parallel region then obtains the thread number of the executing thread and displays it with a `printf` statement.

```

main()
{
  int id;
  omp_set_num_threads(2);
# pragma omp parallel private (id)
  {
    id = omp_get_thread_num();
    printf ("Hello World from OpenMP Thread %d\n", id);
  }
}
```

Using the `dis` command to disassemble the object module produced from the preceding source code results in the following output:

```
       __main_6:  1
0x0:    27bb0001       ldah    gp, 1(t12)
0x4:    2ffe0000       ldq_u   zero, 0(sp)
0x8:    23bd8110       lda     gp, -32496(gp)
0xc:    2ffe0000       ldq_u   zero, 0(sp)
0x10:   23defff0       lda     sp, -16(sp)
0x14:   b75e0000       stq     ra, 0(sp)
0x18:   a2310020       ldl     a1, 32(a1)
0x1c:   f620000e       bne     a1, 0x58
0x20:   a77d8038       ldq     t12, -32712(gp)
0x24:   6b5b4000       jsr     ra, (t12), omp_get_thread_num
0x28:   27ba0001       ldah    gp, 1(ra)
0x2c:   47e00411       bis     zero, v0, a1
0x30:   23bd80e8       lda     gp, -32536(gp)
0x34:   a77d8028       ldq     t12, -32728(gp)
0x38:   a61d8030       ldq     a0, -32720(gp)
0x3c:   6b5b4000       jsr     ra, (t12), printf
0x40:   27ba0001       ldah    gp, 1(ra)
0x44:   23bd80d0       lda     gp, -32560(gp)
0x48:   a75e0000       ldq     ra, 0(sp)
0x4c:   63ff0000       trapb   0x50:   23de0010        lda     sp, 16(sp)
0x54:   6bfa8001       ret     zero, (ra), 1
0x58:   221ffff4       lda     a0, -12(zero)
0x5c:   000000aa       call_pal        gentrap
0x60:   c3ffffef       br      zero, 0x20
0x64:   2ffe0000       ldq_u   zero, 0(sp)
0x68:   2ffe0000       ldq_u   zero, 0(sp)
0x6c:   2ffe0000       ldq_u   zero, 0(sp)
         main:
0x70:   27bb0001       ldah    gp, 1(t12)
0x74:   2ffe0000       ldq_u   zero, 0(sp)
0x78:   23bd80a0       lda     gp, -32608(gp)
0x7c:   2ffe0000       ldq_u   zero, 0(sp)
0x80:   a77d8020       ldq     t12, -32736(gp)
0x84:   23defff0       lda     sp, -16(sp)
0x88:   b75e0000       stq     ra, 0(sp)
0x8c:   47e05410       bis     zero, 0x2, a0
0x90:   6b5b4000       jsr     ra, (t12), omp_set_num_threads
0x94:   27ba0001       ldah    gp, 1(ra)
0x98:   47fe0411       bis     zero, sp, a1
0x9c:   2ffe0000       ldq_u   zero, 0(sp)
```

```
0xa0:   23bd807c        lda     gp, -32644(gp)
0xa4:   47ff0412        bis     zero, zero, a2
0xa8:   a77d8010        ldq     t12, -32752(gp)
0xac:   a61d8018        ldq     a0, -32744(gp)
0xb0:   6b5b4000        jsr     ra, (t12), _OtsEnterParallelOpenMP 2
0xb4:   27ba0001        ldah    gp, 1(ra) :     a75e0000        ldq     ra,
0(sp)
0xbc:   2ffe0000        ldq_u   zero, 0(sp)
0xc0:   23bd805c        lda     gp, -32676(gp)
0xc4:   47ff0400        bis     zero, zero, v0
0xc8:   23de0010        lda     sp, 16(sp)
0xcc:   6bfa8001        ret     zero, (ra), 1
```

1 `__main_6` is the outlined routine created by the compiler for the parallel region beginning in routine `main` at listing line 6. The format for naming the compiler-generated outlined routines is as follows:

`__original-routine-name_listing-line-number`

2 The call to `_OtsEnterParallelOpenMP` is inserted by the compiler to coordinate the thread creation and execution for the parallel region. Run-time control remains within `_OtsEnterParallelOpenMP` until all threads have completed the parallel region.

13.6.2 Debugging and Application-Analysis Tools

The principal tool for debugging OpenMP applications is the Ladebug debugger. Other tools include Visual Threads, the atom tools `pixie` and `third`, and the OpenMP tool `ompc`.

13.6.2.1 Ladebug

This section describes how to use the Ladebug debugger with OpenMP applications. It explains unique considerations for an OpenMP application over a traditional, multithreaded application. It uses the example program in Section 13.6.1 to demonstrate the concepts of debugging an OpenMP application. For more complete information on debugging multithreaded programs, see the *Ladebug Debugger Manual*.

Because OpenMP applications are multithreaded, they can generally be debugged using the same strategies as regular multithreaded programs. There are, however, a few special considerations:

- As with optimized code, the compiler alters the source module to enable OpenMP support. Thus, the source module shown in the debugger will not reflect the actual execution of the program. For example, the generated routines from the outlining process performed by the compiler will not be visible as distinct routines. Prior to a debugging session, an output listing or object module disassembly will provide the names of these routines. These routines can be analyzed within a Ladebug session in the same way as any normal routine.

- The OpenMP standard defines thread numbers beginning with thread 0 (the master thread). Ladebug does not interpret OpenMP thread numbers; it interprets their pthreads equivalent, whose numbering begins with thread 1.
- The call stack for OpenMP slave threads originate at a pthreads library routine called thdBase and proceed through a libots3 routine called slave_main.
- Variables that are private to a parallel region are private to each thread. Variables that are explicitly private (qualified by firstprivate, lastprivate, private, or reduction) have different memory locations for each thread.

To debug a parallel region, you can set a breakpoint at the outlined routine name. The following example depicts starting a Ladebug session, setting a breakpoint in the parallel region, and continuing execution. The user commands are described in footnotes.

```
> ladebug example                         1
Welcome to the Ladebug Debugger Version 4.0-48
------------------
object file name: example
Reading symbolic information ...done
(ladebug) stop in __main_6                2
[#1: stop in void __main_6(int, int) ]
(ladebug) run                             3
[1] stopped at [void __main_6(int, int):6 0x1200014e0]
      6 # pragma omp parallel private (id)
(ladebug) thread                          4
 Thread Name            State        Substate    Policy       Pri
 ------ --------------- ------------ ---------- ------------ ---
>*> 1 default thread    running                  SCHED_OTHER  19
(ladebug) cont                            5
Hello World from OpenMP Thread 0
[1] stopped at [void __main_6(int, int):6 0x1200014e0]
      6 # pragma omp parallel private (id)
(ladebug) thread                          6
 Thread Name            State        Substate    Policy       Pri
 ------ --------------- ------------ ---------- ------------ ---
>*    2 <anonymous>     running                  SCHED_OTHER  19
(ladebug) cont                            7
Hello World from OpenMP Thread 1
Process has exited with status 0
```

1 Start a Ladebug session with the example application.

2 Create a breakpoint to stop at the start of the outlined routine __main_6.

3 Start the program. Control stops at the beginning of __main_6.

4 Show which thread is actively executing the parallel region (pthread 1, OpenMP thread 0, in this example).

5. Continue from this point to allow the parallel region for OpenMP thread 0 to complete and print the `Hello World` message with the proper OpenMP thread number before the breakpoint is hit again.

6. Show the next thread that is actively executing the parallel region (pthread 2, OpenMP thread 1).

7. Continue from this point to print the next message and complete the execution of the program.

The following example shows how to set a breakpoint at the beginning of the outlined routine when pthread 2 (OpenMP thread 1) begins at execution of the parallel region.

```
> ladebug example
Welcome to the Ladebug Debugger Version 4.0-48
------------------
object file name: example
Reading symbolic information ...done
(ladebug) stop thread 2 in __main_6                            1
[#1: stop thread (2) in void __main_6(int, int) ]
(ladebug) r
Hello World from OpenMP Thread 0
[1] stopped at [void __main_6(int, int):6 0x1200014e0]
      6 # pragma omp parallel private (id)
(ladebug) thread
 Thread Name             State        Substate    Policy       Pri
 ------ ---------------- ------------ ----------- ------------ ---
>*    2 <anonymous>      running                  SCHED_OTHER  19
(ladebug) c
Hello World from OpenMP Thread 1
Process has exited with status 0
```

1. Stop OpenMP thread 1 (pthread 2) when it encounters the start of the parallel region.

Debugging the OpenMP combined work-sharing constructs (`for` and `sections` directives) is analogous to the process shown in the preceding examples.

Note

The Ladebug debugger does not yet fully support OpenMP debugging. Variables that are declared as `threadprivate` are not recognized by Ladebug and cannot be viewed.

13.6.2.2 Visual Threads

Programs instrumented with OpenMP can be monitored with the Compaq Visual Threads (`dxthreads`) product, which is on Associated Products Volume 1 CD-ROM. For details, see the Visual Threads online help.

13.6.2.3 Atom and OpenMP Tools

OpenMP applications can be instrumented using ompc, a special tool created for monitoring OpenMP applications, and the Atom-based tools pixie (for profiling an executable) and Third Degree (third, for monitoring memory access and potential leaks).

The ompc tool captures the pertinent environment variables settings and maintains counts of calls to the OpenMP-related run-time library routines. It generates warnings and error messages to call attention to situations that may not be what the developer intended. Finally, based on the settings of environment variables, it will trace all calls to these run-time library routines and report, by OpenMP thread number, the sequence in which they were called. See ompc(5) for more information.

The Atom-based pixie tool can be used to detect inefficient thread usage of the application. As described in Section 13.6.1, slave threads will wait, or spin, until a new parallel region is encountered or the MP_SPIN_COUNT expires. If an application experiences long delays between parallel regions, the threads will spin until they are put to sleep. By instrumenting the application with pixie, you can see where the application is spending most of its time. If your application is spending large amounts of compute time in slave_main, this is a good indication that the threads are spending too much time spinning. By reducing the MP_SPIN_COUNT (the default is 16000000) for these types of applications, you may realize better overall performance. For more information about pixie, see Chapter 8 and pixie(1).

For information about the Third Degree tool, see Chapter 7 and third(1).

13.6.2.4 Other Debugging Aids

Other debugging aids include the following:

- The compile-time option -check_omp, which embeds additional run-time checking to detect deadlock and race conditions (see Section 13.1).
- The omp_set_num_threads and mpc_destroy functions, which let you modify the number of active threads in a program.

You can modify the number of active threads in a program by calling either omp_set_num_threads or mpc_destroy. In either case, any data declared threadprivate and associated with the slave threads is reinitialized to the values at application startup. For example, if the active number of threads is 4 and a call is made to set the number to 2 (via omp_set_num_threads), then any threadprivate data associated with OpenMP threads 1, 2, and 3 will be reset. The threadprivate data associated with the master thread (OpenMP thread 0) is unchanged.

For more information about `mpc_destroy`, see Section 13.3.2.

14

Posting and Receiving EVM Events

This chapter describes the user-level programming interface to the Tru64 UNIX Event Manager (EVM). The principal issues addressed in this chapter are as follows:

- Determining which status changes should be established as events
- Designing the contents of events
- Using EVM's API functions and utilities to post, subscribe to, and access events

In the context of this chapter, an event is an indication that something has happened that may be of interest to one or more of the following:

- A system administrator, application manager, or some other class of user
- System monitoring software
- The operating system
- An application program

Interested parties may be part of the local system or they may be running on a remote system.

This chapter addresses only user-level event handling. For information about posting and subscribing for events within the kernel, see `kevm`(7).

This chapter provides information about the following topics:

- Events and event management (Section 14.1)
- An overview of how EVM events are handled (Section 14.2)
- Starting and stopping EVM (Section 14.3)
- Authorization to post and access events (Section 14.4)
- Contents of an EVM event (Section 14.5)
- Designing a set of events (Section 14.6)
- The EVM programming interface (Section 14.7)
- Adding an event channel to EVM (Section 14.8)

Details about the functions supported by the EVM programming interface are provided in online reference pages. (Section 14.7 provides summary descriptions and examples of some of the most commonly used EVM functions.)

The event viewer and the command-line interface are not addressed in this chapter. For information about the viewer, see the online help; for information about command-line operations, see the *System Administration* manual.

14.1 Events and Event Management

EVM provides a centralized means of posting, distributing, storing, and reviewing event information — regardless of the event channel used by individual event posters and without requiring existing posters to change how they interact with their current channels. EVM makes event information more accessible to system administrators than was possible in earlier versions of the Tru64 UNIX system, and it provides a flexible infrastructure that can be used as an event distribution channel by:

- Tru64 UNIX development groups
- Independent software vendors
- Customer-application developers
- Other event channels

The mechanism used to pass event information is known as an **event** (or event notification), and the component generating the event is known as the **event poster**. EVM's event-posting mechanism is a one-way communication channel. It is intended to allow the poster to communicate information to any entity that cares to access it. The poster does not need to know which entities, if any, are interested in accessing an event being posted.

An entity that expresses an interest in receiving event information is known as an **event subscriber**. Depending on the event, subscribers might include system administrators, other software components, or ordinary users. Some events may never be subscribed to.

Events can be posted and subscribed to by any process, and the same process can be both a poster and a subscriber. However, in all cases, the ability to post and access specific events is governed by security authorizations (Section 14.4).

Previous versions of the Tru64 UNIX system supported a variety of disparate channels for handling events, some standard and some proprietary. In the simplest case, an **event channel** is a static ASCII log file containing event information, from a single source, that a user can view by means of standard UNIX tools (for example, `more`). Examples of more active channels include the system logger (`syslog`) and the binary event logger (`binlog`), both of which use daemon processes to receive, log, and forward event information from multiple sources. (See `syslogd`(8) and `binlogd`(8) for information about `syslog` and `binlog`.)

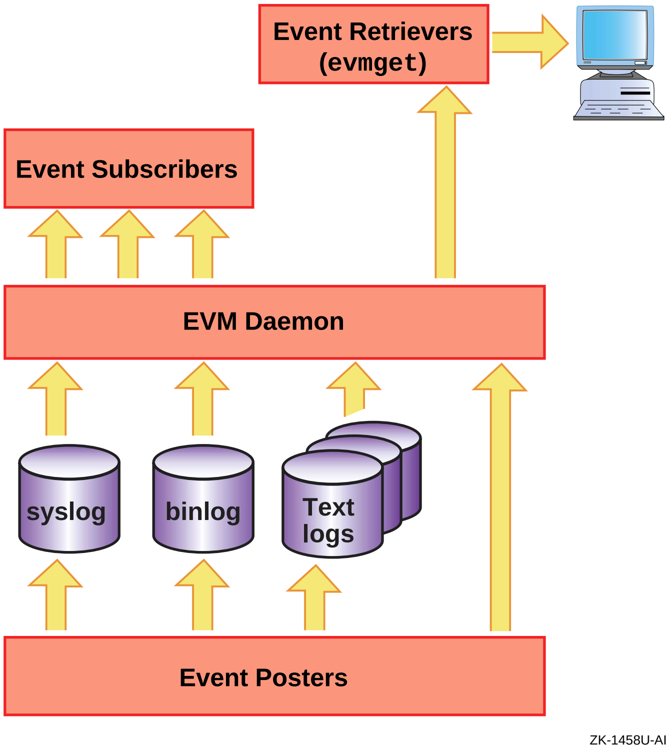
EVM provides a single point of focus for multiple event channels by combining events from all sources into a single event stream. Interested parties can either monitor the combined stream in real time or view historical events retrieved from storage. EVM's viewing facilities include a graphical event viewer, which is integrated with the SysMan application suite, and a full set of command-line utilities, which allow events to be filtered, sorted, and formatted in a variety of ways. EVM can also be configured to perform automatic notification of selected conditions.

Note

EVM is a facility for broadcasting messages and should not be used to implement a private point-to-point communication channel between two processes. Using EVM for such purposes could have a negative impact on system performance. If you need to establish communication with another process and the information you will be sending to that process is not of interest to system administrators or to any other processes, you should use a more direct communication channel, for example, sockets or pipes.

Figure 14–1 provides an overview of posting, subscribing, and retrieval operations. (See `evmget`(1) for information on event retrieval operations.)

Figure 14–1: EVM Overview



14.2 Overview of How EVM Events Are Handled

An EVM event is a package of data that can be passed between various software components and stored for later review.

In order to be posted, an event must match an **event template** that is stored in an EVM database. Data items for an event can be supplied in either the posted event itself or the matching template. The contents of the event received by the event subscriber result from merging the data items in the posted event with those in the event template. See Section 14.6.3.2 and Section 14.6.3.3 for more information about matching and merging.

A typical event life cycle involves the following operations:

1. Templates are created for all events that some process may want to post, and these templates are stored in an EVM database when the product or subsystem is installed.
2. Processes that are interested in receiving events establish a connection to the EVM daemon and then issue subscription requests, specifying a filter that identifies the set of events in which they are interested.

3. When a process or kernel component detects an eventworthy status change, it connects to EVM and posts the event that corresponds to that status change.

   Alternatively, if the event is posted to an event channel other than EVM, the channel processes the event in its own way, which may include logging it. The channel may then convert the event to EVM format and pass it to EVM.

4. EVM validates the request to post the event, checking, among other things, whether the event has a corresponding template in the EVM template database and whether the poster is authorized to post the event. If it is valid, EVM creates a merged version of the event by combining data items in the posted event with data items in the template.

5. EVM passes the merged event to all processes that have subscribed to receive the event.

6. Subscribers handle the event in some appropriate way; for example, they could store it, send mail to the system administrator, or initiate an application failover.

14.3 Starting and Stopping EVM

EVM is automatically started at system startup time. Similarly, it is automatically stopped when the system is shut down.

See the *System Administration* manual for information about starting and stopping the EVM daemon.

See Section 14.6.3.4 for information on how EVM establishes the event template database at startup and for information on how to subsequently modify that database.

14.4 Authorization to Post and Access Events

Security is an important consideration when dealing with the posting and accessing of events:

- Uncontrolled access to certain event information could provide an unauthorized user with sensitive information about system operation.
- Uncontrolled posting of certain events could allow an unauthorized user to initiate critical system actions, for example, a system shutdown.

The system administrator can control who is allowed to access and post events by modifying the EVM authorization file, `/etc/evm.auth`.

See the *System Administration* manual for additional information about access control.

14.5 Contents of an EVM Event

The format, type, and purpose of the data carried in events varies widely. In a simple case, an event is a simple text message. In a more complicated case, an event might be a complex set of binary data that requires an understanding of its format to be able to translate it. For two cooperating applications, an event might be a simple binary flag or count. Note, however, that an event is not required to carry data. In some cases, transmitting only a particular event type from poster to subscriber may be all that the subscriber needs to recognize that a particular status change has occurred.

Event data is complicated by the need to be able to translate it in various ways. For an event to be human readable, it must be possible to combine numeric data with explanatory text. At the same time, the information in the same event may need to be seen as binary data by a subscribing process. In addition, it may also be necessary to have the human-readable form in multiple languages.

The EVM data mechanism allows data to be carried in any way that is appropriate to the event. The constructs that provide this capability are the EVM format data item (one of EVM's standard data items) and EVM variable data items.

Event data structures contain two types of data items:

- Standard data items — A fixed set of items with predefined names.
- Variable data items — Items whose names and types are typically defined by the event designer.

When you create an event, you can include any number of data items. When you post the event, EVM automatically adds certain environmental standard data items such as the host name and the timestamp.

14.5.1 Standard Data Items

Standard data items are data items that are commonly required in an event and are understood (and may be acted upon) by EVM. The names of standard data items are specified as enumerated constants; consequently, the names themselves do not take up a significant amount of space in an event.

Some standard data items are inserted in an event by the application that posts the event or by the template for the event. Others are automatically inserted, as needed, by EVM components. You can extract any data item from an event.

The standard data items are described in `EvmEvent`(5). Items that require special attention are listed in Table 14–1 and are described in the sections that follow.

Table 14–1: Standard Data Items

Data Item	Description
Event name (Section 14.5.1.1)	Names the event.
Format (Section 14.5.1.2)	Specifies the event message string.
Priority (Section 14.5.1.3)	Indicates the likely importance of the event to its subscribers. Does not affect the order of event distribution.
I18N catalog (Section 14.5.1.4)	Name of the I18N catalog file for internationalized events.
I18N msgset ID (Section 14.5.1.4)	Identifies the message set within the I18N message catalog.
I18N message ID (Section 14.5.1.4)	I18N message ID for event format.
Cluster Event (Section 14.5.1.5)	In a TruCluster environment, this item causes the event to be distributed to all nodes of the cluster.
Reference (Section 14.5.1.6)	Provides a reference to the event explanation text.

The following sections provide detailed descriptions of the standard data items defined in Table 14–1.

14.5.1.1 Event Name Data Item

The **event name** data item is a character string that identifies an event, generally indicating the origin of the event and what happened. An event name uniquely identifies a particular type of event, and different instances of the same event have the same name.

EVM uses the event name to decide if a user is authorized to post or access a particular event and to find template information for the event. Client applications can use event names in conjunction with an **event filter** (Section 14.7.11) to select the events that they want to receive, and to decide on any action that they need to take when an event arrives. The system administrator can use event names to search the event logs for events that are of particular interest.

An event name has the following characteristics:

- A name is series of one or more components, which are separated by dots.
- A component is made up of one or more letters, digits, and underscore characters, in any order.
- There is no restriction on the number of components in a name.
- A name cannot begin or end with a dot.

An event cannot be posted unless it contains an event name with at least three components.

The components of an event name represent an event hierarchy, beginning with the first (left-most) component. The first component is the most general, and identifies, at a global level, the entity that is responsible for posting the event. Order the remaining components so that each adds a further level of detail, including the component that determined that an event needed to be posted, and information about what happened, with the last component giving the most specific information. For example, in the event name sys.unix.fs.filesystem_full:

- sys.unix indicates that the event is a system event, posted by a component of the operating system.
- fs indicates that the event is posted by the file system component.
- filesystem_full indicates that the incident being reported is that a file system is full.

A basic event name that indicates what happened can be extended when the event is posted, by adding more components to give additional information. You can extend the preceding sample event name with information about which file system is full by appending the name of the file system as an extra component, for example, sys.unix.fs.filesystem_full.usr. Although the name has been extended, the event is still a sys.unix.fs.filesystem_full event because all of the components of the shorter name match the extended name. A search for an event named sys.unix.fs.filesystem_full will still find the event with the extended name.

The naming scheme provides an open-ended way to identify events, which allows you to provide detail to any level. Careful naming helps subscribers select specific events or event classes for viewing and monitoring: the more detail you put in the event name, the more precise the specification criteria can be. For example, given the event name myco.myprod.env.temp.ok and myco.myprod.env.temp.high, a system administrator can monitor all temperature-related events by specifying myco.myprod.env.temp or restrict monitoring to events that report a high temperature by specifying myco.myprod.env.temp.high.

To avoid confusion and possible incorrect action by the system administrator or monitoring software, make sure that you give a unique name to each new event. You can avoid duplicating a name that is already in use by following these established conventions:

- For events that are posted by third-party products or applications, make the first three name components identify the vendor of the product, the product, and the program or product component that posts the event. To make sure that the vendor component is globally unique, you should generally avoid choosing two or three initials for this component — instead, use the full company name or an abbreviation that no other company is likely to choose.

- If you are a user application developer or administrator, generally follow the same conventions that we recommend for product vendors, identifying the organization in the first component. However, you can also choose to use the prefix `local` because this prefix should never clash with events that are posted by any externally-developed products that you might install.

  Make the second component identify the application or group that is responsible for posting the event; for example, your events could begin with the components `myco.payroll` or `myco.admin`.
- The first two components of all events that are posted by Tru64 UNIX are `sys.unix`, and the third component identifies the subsystem that posts the event. The use of `sys` as a first component is reserved for system events.

The following table lists names that have specific meanings. To prevent name clashes, do not use these names as the first component of events, except as specified.

Name	Use	Examples
`sys`	Operating system events	`sys.unix.evm.logger.config_err`
`local`	Local user organization use	`local.app.payroll.started` `local.app.payroll.started`
`test`	Available for local test event posting	

After you have chosen and published an event name, avoid changing it because the events that an application posts are part of its external interface, and other applications or product components might depend upon receiving them.

14.5.1.1.1 Reserved Component Names

As stated in the previous section, the more detail you include in an event name, the easier it is for an administrator to use an EVM filter to find particular types of events. For example, the event `sys.unix.hw.registered.cpu` reports that the operating system's hardware management subsystem has detected and registered a processor. The event contains the hardware ID of the processor in a variable data item (Section 14.5.2), and you can see the hardware ID if you display the event with the event viewer or `evmshow`. If you are tracing a problem that is related to that processor, you might want to use an event filter to select just the events that include that hardware ID. You cannot filter events by the values of the variables that they contain, so you can only make this selection if the name of the event also includes the information that you want to select. In this case, the hardware management subsystem makes the selection possible by extending the name with the hardware ID that is assigned to each device that it registers. If the hardware ID is 2, you could search for related events with the following command:

```
evmget -A -f "[name sys.unix.hw.*.2]"
```

In the example, the asterisk (`*`) wildcard character ensures that you find all events posted by the hardware subsystem that include the identifier 2, regardless of what they are reporting. However, this filter might not select all of the events for the device that interests you because subsystems other than the hardware subsystem might also report information about the processor, and you want to include those events in the search. You can increase the scope of the search by specifying `[`*`name`*` *.2]`, but other events in the log with the number 2 in their names that have nothing to do with hardware IDs would be included in the results.

The convention of **reserved component names** makes it possible to select related events, regardless of which subsystem or application posts them. By convention, a reserved component name begins with an underscore character (_) and always identifies a particular type of entity. The posting subsystem or application appends the reserved component name to the base event name before posting the event, and, depending on the entity being identified, the following component might then identify the specific instance.

Although reserved component names are generally defined on behalf of a particular subsystem or application, after a name has been defined, its use is not restricted — anything that has information to report about the entity can include the reserved name and entity identifier in the events that it posts. For example, the reserved component name `_hwid` specifies a hardware device identifier, as defined by the hardware management subsystem, and must be followed by the device-id. So, continuing the previous example, the name of the event that reports the registration of the processor is `sys.unix.hw.registered.cpu._hwid.2`. You can find all events that relate to all hardware devices with the following command:

```
evmget -A -f "[name *._hwid]"
```

You can then narrow the search to the processor that most interests you with the following command:

```
evmget -A -f "[name *._hwid.2]"
```

The use of the wildcard in the filter for all components except the hardware ID components ensures that the search yields all related events, regardless of which subsystem or application posted them, provided that those posters followed the convention.

The reserved component name convention requires that, when a reserved component name is used, the posted event must contain a variable with the same name (including the leading underscore) and value. This convention allows a subscribing client to retrieve the value from the event through EVM's normal API functions, rather than having to parse the event name to find the value. Because the convention would otherwise require the same information to be added to the event twice, the `EvmEventPost( )` function automatically finds any variables in an event whose names begin with an underscore, and, if the calling program has not already included corresponding components in the event name, it appends them before posting the event. See `EvmEventPost`(1) for more information about automatic name extension.

An event can include any number of trailing reserved component names, in any order and in any component position following the event's base name. Each reserved component must be followed immediately by its associated value component.

You do not need to include a reserved component in an event's template name when registering the event because the EVM name matching scheme finds the best match for a posted event. In the example shown previously in this section, the template (and the base name of the event) would be `sys.unix.hw.registered`. Reserved components are appended to the name to give a further level of detail when the event is posted.

The convention of defining reserved component names that begin with a single underscore is reserved for system use. For local sites and third-party product vendors, establish your own conventions using a double underscore to start reserved component names, for example, `__prod_code`.

For a list of the reserved component names used by Tru64 UNIX, see `EvmEvent`(5).

14.5.1.1.2 Comparing Event Names

Use EVM's API functions to compare the name of an event with the name you are looking for. Do not use `strcmp( )`; the name of the incoming event may have more components than you expected. See Section 14.7.12.9 for more information on event name matching.

14.5.1.2 Event Format Data Item

The format data item is a human-readable character string that summarizes the event. The format line will be expanded and displayed in the following situations:

- When the event is displayed using the `evmshow` command
- When the event is displayed using the event viewer

- When a program calls a display API function such as `EvmEventFormat( )`

The format data item can include references to other data items contained in the event. Consider the following examples:

```
Application close-down has started
```

```
Close-down of application $app has started
```

In the second example, `$app` is replaced by the name of the application, provided that the event contains a variable value with that name — so the line might be displayed as:

```
Close-down of application payroll has started
```

Variables are explained in Section 14.5.2.

A format line should begin with a brief and consistent identification of the posting component, and it should be a concise indication of what has transpired. Consider the following guidelines:

- You do not need to write a full sentence, but make sure that what you write can be clearly understood.
- Do not write multiple sentences and do not include a period at the end of the message. (Use the event's explanation text to provide the full (and more grammatical) description of what happened. See Section 14.6.2 for information on how explanation text is handled.)
- Try to place the most important information — the phrase that tells the administrator what happened — at the beginning of the line to make it unnecessary for the administrator to scroll the viewer window horizontally to find this information. Variable data is often better placed towards the end of the line.
- You do not have to include a reference to every variable in the format string. The administrator can easily get the variable data by requesting a detailed description of the event. However, try to include the key data, such as the name of a file system or device.

You can include variable names for substitution by preceding them with the `$` character, for example, `$app`. If necessary, you can include standard data items by preceding their names with the `@` character, for example, `@host_name`. You can specify standard data item names in upper– or lowercase, but variable names are case sensitive.

You can escape the special meaning of `$` and `@` characters in the format text by preceding them with a backslash (\). To include a literal backslash in the text, use a double backslash (\\). To separate the name of the data item or variable from adjacent text, you can surround the name in braces, for example, `${app}`.

You can control the display of a variable or data item by following its name by a percent sign (%) and a *format specifier*. The format specifier can include the minimum width of the display field and the display format. For example, the following format string specifies that the variable named `device_id` should be displayed in hexadecimal notation and the `temperature` should be displayed as a floating-point number with 2 decimal places:

```
The cabinet temperature for device 0x$device_id%x is $temperature%.2 degrees
```

The corresponding event might be displayed as:

```
The cabinet temperature for device 0x7a is 62.4 degrees
```

For a full list of valid format specifiers, see `evmshow`(1).

Table 14–2 contains some examples of the way that variables and data items are substituted into an event's format text to produce a formatted version of the event.

Table 14–2: Substituting Variables into Event Text

Format Data Item	Variable Data Item	Resulting Text
`Application started`	None	`Application started`
`Debug message:  $msg`	`msg (string) = "Check-point reached"`	`Debug message:  Checkpoint reached`
`Temperature too high:  $temp`	`temp (float) = 87.2`	`Temperature too high:  87.2`
`This event was posted by @user_name`	None	`This event was posted by jem`

14.5.1.3 Event Priority Data Item

The priority of an event is used as the basis for selecting events for logging, sorting, reviewing, and taking action — human or automated. The priority is not used to establish the order in which events will be delivered to subscribing EVM clients; events are delivered in the order in which they are received by the EVM daemon. The priority is an integer value in the range 0-700, with 0 being the least significant priority. See `EvmEvent`(5) for information about the various priority levels.

You can expect that system administrators will respond to event priorities in the following ways:

- Configure the logger to log all events with priority 200 (notice level) or higher.
- Configure the logger to send mail or issue an alert (for example, page someone) when an event of priority 600 (alert level) or higher is posted.
- Use the viewer to search for all events with priority 300 (warning level) or higher.

- Use the viewer to peruse logged events of all priorities to help analyze problems or to verify that the system is operating correctly.

Do not hardcode the priority into the posting code; it should always be established in the template file.

When choosing the priority of an event, try to be consistent with other events in your own application and other similar applications. Do not try to use the full range of possible priorities if they are not applicable to your application. Use the priority information in `EvmEvent`(5) as a guideline, and look at existing templates to see what priorities are being used by similar events.

Choose the priority of an event carefully and objectively. Think about what you are reporting, not the possible consequences. For example, some application failures are critical, but many are not. The determination of which applications are critical should be left to a higher-level component. The higher-level component can subscribe to error events and issue a critical-level event if it is notified of an application error that it knows to be critical.

Based on these considerations, events that indicate an error situation but do not necessarily meet the critical criteria should be given a priority of 400 (error level), not 500 (critical level). On the other hand, an event reporting an excessive system temperature is almost always critical.

Event priorities for events that are related to each other should be established independently of each other. This means, for example, that if you post a failure event with a priority of 500 to indicate that an application has failed and you later post a related event to indicate that is has been restored, the second event should have a lower priority, for example, 200. Giving the restore event a priority of 500 may cause an action that is not appropriate for the event.

14.5.1.4 I18N Catalog Name, Message Set ID, and Message ID Data Items

If your events contain text that is likely to be viewed by administrators in different countries, consider internationalizing them. To internationalize your events, you need to supply an I18N catalog file containing the format strings for all of the events. Note that you should still include the text within the event, in case the catalog file is not available when the event text is viewed.

The catalog file should be located according to normal I18N rules, and you must include the name of the file in your event specifications. You can provide different versions of the catalog file for different language locales. See the *Writing Software for the International Market* manual for more information.

You can optionally break the catalog file into multiple message sets and specify the message set ID in your events. Note that all messages pertaining to a particular event must belong to the same set.

The data items `I18N_catalog` and `I18N_msgset_id` define the catalog name and, if applicable, the message set for the format text and any string variables contained in the event and having an associated message ID. The `I18N_format_msg_id` data item gives the message ID for the format text. If no catalog or message ID is supplied, the format text specified in the format data item is used to display the event summary.

See Section 14.6.4 for information on how to establish translations for event text.

14.5.1.5 Cluster Event Data Item

If the cluster event data item is set to true and the event is posted on a system that is a member of a TruCluster system, EVM distributes the event to subscribers on all active nodes of the cluster. If the item is not set, or is set to false, the event is only distributed to subscribers on the local node. This item has no effect if the event is posted on a standalone system. Events that have the cluster event flag set to true are referred to as cluster-wide events.

When used in a template or an event specification to `evmpost`, the value of the cluster event data item is true or false. When it is used a a data item in an API function, its value is `EvmTRUE` or `EvmFALSE`.

Cluster-wide events differs according to whether the event is posted from user level or from the kernel:

- At user level, a cluster-wide event is posted to the local EVM daemon in the same way as any other event, and the daemon then distributes the event to its own subscribers and to the EVM daemons running on the other active nodes. The event_id data item is set by the local daemon and is not modified by the other daemons before it is distributed to their subscribers. This means that:
  - Subscribers on all nodes receive identical copies of the event.
  - The event_id data items of the copies received on the non-posting nodes will most likely be out of sequence with the events that were posted locally on those nodes.
- Cluster-wide events posted from the kernel are distributed to the other nodes without first being passed through the local daemon, and consequently they are not given a local event_id value. In this case, the daemons running in each node recognize that the event_id value is not set, and sets the value itself. This means that:

- Subscribers on different nodes of the cluster receive copies of the same kernel-posted cluster-wide event with different event_id values.
- The event_id data items of the copies received on the non-posting nodes will be in sequence with the events that were posted locally on those nodes.

For detailed information about posting and subscribing for kernel-level events, see `kevm`(7).

EVM ensures that subscribers that are connected to the cluster through the cluster alias will receive only a single copy of a cluster-wide event. However, the EVM logger subscribes for and logs all events that match its filter, regardless of which cluster member posted them, so a separate copy will be logged on each node. Consequently, if the `evmget` command is used to retrieve stored events by connecting to the cluster alias, one copy of the event will be returned for each node that logged it.

If you are developing a cluster-based application you should consider carefully which events — if any — need to be designated as cluster-wide events. It is especially important that intended subscribers are designed to recognize which node posted each instance of the event and to take appropriate action. You can use the member _id data item in conjunction with standard library functions such as `clu_get_info( )` to obtain details of the posting and local node. See `EvmEvent`(5) and `clu_get_info`(3) for more information.

14.5.1.6 Reference Data Item

The reference data item is used to find the event explanation text for a detailed display of the event. The value of this item is passed to the event channel's explain script, along with the name of the event, to allow it to identify the explanation text associated with the event. Because each channel can have its own explain script, the format of the field might be different for each channel — however, for events that are stored in and retrieved from the EVM log (the evmlog channel), this item should contain a string of the following form:

```
cat:catalog_name[:set_number]
```

The *`catalog_name`* is the name of an I18N catalog containing the explanation text for the event. To allow the explain script to locate the appropriate message, each explanation message in the catalog must begin with the name of the event enclosed in braces.

The optional *`set_number`* is the number of the catalog message set containing the explanation. If no set number is supplied, the whole catalog is searched.

Third-party product vendors and local applications should not add explanations to the Tru64 UNIX explanation catalog, evmexp.cat, but should provide separate catalogs. The value of this item should generally be set as a global data item in the template file.

See Section 14.6.2 for details about how to write event explanation text.

14.5.2 Variable Data Items

You can use variable data items in your events to provide any information that is different for each instance of an event. For example, if you are posting an event because you have detected a high temperature, you could include the actual temperature as a floating-point value in a variable data item.

Variable data items have the following properties:

- Name
- Type
- Value
- Size (implicit for most types)
- I18N message ID (optional — applies to string variables only)

A variable name can be made up of any combination of uppercase or lowercase alphanumeric characters and the underscore (_) character. Names should be meaningful, but remember that the variable name is carried inside the event, and the longer the name, the larger the physical event.

A variable data item can be extracted and used directly by the subscriber or combined with the event format text string to produce a formatted version of the event for display.

Table 14–3 lists the variable types that EVM supports.

Table 14–3: EVM's Variable Data Types

Type Identifier	Size and Type
EvmTYPE_BOOLEAN	8-bit integer
EvmTYPE_CHAR	8-bit character
EvmTYPE_INT16	16-bit signed integer
EvmTYPE_INT32	32-bit signed integer
EvmTYPE_INT64	64-bit signed integer
EvmTYPE_UINT8	8-bit unsigned integer
EvmTYPE_UINT16	16-bit unsigned integer
EvmTYPE_UINT32	32-bit unsigned integer

Table 14–3: EVM's Variable Data Types (cont.)

Type Identifier	Size and Type
EvmTYPE_UINT64	64-bit unsigned integer
EvmTYPE_FLOAT	32-bit floating-point value
EvmTYPE_DOUBLE	64-bit floating-point value
EvmTYPE_STRING	Null-terminated character string
EvmTYPE_OPAQUE	Binary data whose size must be specified explicitly

In general, because variables contain information that is specific to an instance of an event, they should be included in the event by the poster. However, for documentation purposes, it is also useful to include variable names and types, along with dummy values, in the template version of the event. See Section 14.6.3 for information on templates.

14.6 Designing a Set of Events

When designing an application or subsystem, you also need to design an associated set of events.

EVM events need to be designed with care. Events must meet the requirements of two styles of interfaces: the human style (readable text) and the program style (binary data). After an event is posted, it can be seen and acted upon in either its text form or its binary data form.

Designing an event involves the following considerations:

1. Decide on a family name for a set of related events. (See Section 14.5.1.1 for details.)
2. Create a list of the status changes that might be of interest to a monitoring entity, and choose a name for each event. (See Section 14.6.1 for details.)
3. Decide on the contents of each event. Each individual event must have a unique name. Most events need a format string and a priority, and many also need variables. For each variable, consider the type and possible values. (See Section 14.5 for details.)
4. Write a detailed description of each event for documentation purposes. Include details about what the event means, when it might occur, any actions the user or responsible subscriber should take in response to it, and the contents of the event (particularly any variable data items). The explanation text is usually held in a catalog file and can be accessed for display. (See Section 14.6.2 for details.)

5. For each event, decide which items go into the template and which will be coded by the poster. Except for the event name, all items in both posting code and templates are optional. If an optional item is declared in both places, the poster's declaration has precedence. (See Section 14.6.3 for details about templates, Section 14.5 for details about event items that are commonly posted, and Section 14.6.3.3 for details about how data items in templates and posted events are merged.)
6. Decide whether the events should be internationalized. If so, choose a name for the I18N catalog file and establish any required message sets within the catalog. (See Section 14.6.4 for details.)

Designers must recognize that an EVM event is an interface upon which other programs or subsystems may rely. Therefore, once established, the interface should generally not be changed.

14.6.1 Deciding Which Status Changes Are Eventworthy

The importance of an event can vary from application to application, and it may be difficult in some cases to decide whether a status change recognized by your code is important enough to communicate to others.

We recommend that you post events for the following types of occurrences:

- When a component makes a change to the system. For example, a system management program might post an event if it adds a new user to the system or changes the network configuration.
- When a potentially significant error occurs. For example, a system management program should post an event if it finds that a key system file is missing, and a device driver should post an event if it detects a disk failure.
- When a boundary indicating that a failure is likely to occur is passed. For example, file system management software might post a warning event when it detects that a file system has passed a boundary and become at least 95 percent full. Take care, however, to avoid posting repeated events of this nature if the state oscillates around the boundary. Your code should typically only post the same event again if the condition is still true after some preset time interval has elapsed, even if the state has dropped below the boundary and recovered several times during the time interval.
- When a user is granted a privilege or takes some action that affects the operation of the system. For example, a system management program might post an event when a disk is mounted or unmounted, or when the system is being closed down.

Do not post events for the following types of occurrences:

- In response to an error made by a user in a session that is under your control and where you have direct communication with the user. For example, a configuration program should not post an event just because a user responded incorrectly to a prompt.
- If you are dealing with an “error” that meets the following criteria: it is expected and is normal behavior, you know why it is happening, and it will not cause a system administrator to take any action.
- If the condition that you have just detected was reported very recently, and reporting it again will serve no useful purpose.

If you do post an event, avoid posting the same event repeatedly over a short period of time if, for example, a condition oscillates between true and false. In some cases, it may be helpful to post an occasional summary event, stating, for example, that the same incident has occurred multiple times within a specified length of time.

High levels of event activity can cause the loss of events because an application may not be able to handle the message load. See Section 14.7.12.10 for information on how to handle missed events.

14.6.2 Writing Event Explanation Text

You should supply explanation text for every EVM event. Explanation text is not included within an event when it is posted, but might be held in a catalog file and referenced by the contents of the event’s reference and name data items (see Section 14.5.1.6). The explanation text for `sys.unix` events is physically held in a catalog file named `evmexp.cat`. To display the explanation text for an event, use the `-x` or `-d` option of `evmshow` or request a detailed view of the event through the SysMan event viewer.

Your explanation should include the name of the event and a description of what it means. If an event can mean different things, depending on the context (for example, the number of occurrences within a given time period or the presence of certain other events), then state that fact and provide a couple of sample situations. Whenever an action is required, state what the action is. If the action varies with the context, then state that fact and provide examples. If the event does not require an action from the user, it is often helpful to state that explicitly.

Example 14–1 shows sample explanation text for a system event.

Example 14–1: Sample Event Explanation Text

```
Example 1:

EVENT sys.unix.evm.daemon.event_activity
```

Example 14–1: Sample Event Explanation Text (cont.)

```
Explanation:

This high-priority event is posted by the EVM daemon when it
detects a high number of events occurring over several minutes.

Action: Use the EVM event viewer or the evmget(1) command to
review the event log for the source of the activity. If the
log does not show high activity around the time at which this
event was posted, it is likely that the events were low priority,
and hence were not logged. You can monitor low-priority events by
running the evmwatch(1) command with an appropriate filter, or
by temporarily reconfiguring the EVM logger to log low-priority
events.

Note: You can change the parameters that control the posting
of this event by modifying the daemon configuration file,
/etc/evmdaemon.conf.
```

14.6.3 Designing Event Templates

Each posted event must have a template, and each template can support one or more events. A template defines the event name and any constant data item within the event.

An event template is used for two purposes:

- To register the event with EVM. An event that is not registered cannot be posted.
- To allow data items that are common to most or all instances of an event (for example, message catalog information) to be held in the template for that event. Including constant data items in a template makes it easy to update this information and relieves event posters from the need to provide this information repeatedly in their post requests. See Section 14.6.3.1 for information on how to decide what to include in an event template.

  See Section 14.6.3.3 for a description of the way that the contents of event templates are merged with the contents of a posted event.

Templates are stored in files that are held in a central database (see Section 14.6.3.4). You can put any number of event templates in a template file.

14.6.3.1 Deciding What to Put in an Event Template

Deciding which items should be supplied by event posters and which should be supplied in the template is a design-level decision. As a general rule, including constant data items in the event template is better than having them hardcoded into an event by posting programs.

A key benefit of the event template mechanism is that it allows you to centralize the constant attributes of all events in your application, making it very easy to change them during the development cycle and providing a single place to look for the information after the application has been deployed.

As a general rule, try to minimize the amount of event information that you hardcode into a posting application program and to put as much as possible into the event template. Typically, your application should only provide:

- Event name (required — it must have at least three components and be at least as long as the event name in the matching template)
- Values of any variables

The template should generally include:

- Event name (required — it must have at least two components)
- Priority
- Format text
- I18N message catalog information (for internationalized events)
- Cluster event flag (if applicable)
- Variables (usually initialized to zero values or empty strings)

Although the posting application is generally expected to supply values for variable data items when it posts an event, including the variable in the template is helpful as well because this makes the template more valuable as a point of documentation. In the template, you should typically give variables zero values (or for string variables, an empty string). In some special cases, providing a real default value in a template (which can be overridden by the poster) may be useful — if this is the case, be sure to describe the situation in a comment in the template file.

The following source file is an example of an event template file containing one event:

```
# Example event file
priority 200           # Default priority
ref  cat:myapp_exp.cat  # Global reference
event {
        name            myco.myapp.env.temperature
        format          "Temperature is $temperature"
        var { name temperature type FLOAT value 0,0 }
}
```

You can include as many variables as you like in an event, but note that opaque variables (binary structures) are not supported in templates.

14.6.3.2 Matching the Names of Posted Events with Event Template Names

Each time an attempt is made to post an event, EVM looks in its template database for the template whose event name matches the name of the posted event. If no match is found, EVM returns an error code to the program that posted the event. If a match is found, EVM then combines the data items held in the template with the items supplied by the posting program. This operation produces the **merged event** that is distributed to subscribers. See Section 14.6.3.3 for details about the merging operation.

The template-matching process requires a match only between the leftmost components of a posted event's name and all of the components of a template event's name. EVM looks for the best match in its database, using the following rules:

- The closest match is the template event whose name exactly matches the most components of the posted event, when compared from left to right.
- A match is considered to have occurred if the posted event has at least as many components as the closest database entry, but not if it has fewer components.
- Components must match exactly.
- At least two components are required in the template name, and at least three are required in the name of the posted event.

It is recommended that you provide a separate template for each distinct event your application will post because this allows you to centralize the event's unique information by storing it in the template. However, the benefit of the best-match mechanism is that it allows you to extend an event's name with different instance information each time you post it; for example, you could add a device name or temperature value as additional components. Having these additional instance components makes it easier to filter and sort events. See Section 14.5.1.1.1 for examples of ways to extend an event's name.

Table 14–4 contains some examples of event name matching between event templates and posted events.

Table 14–4: Name Matching Examples

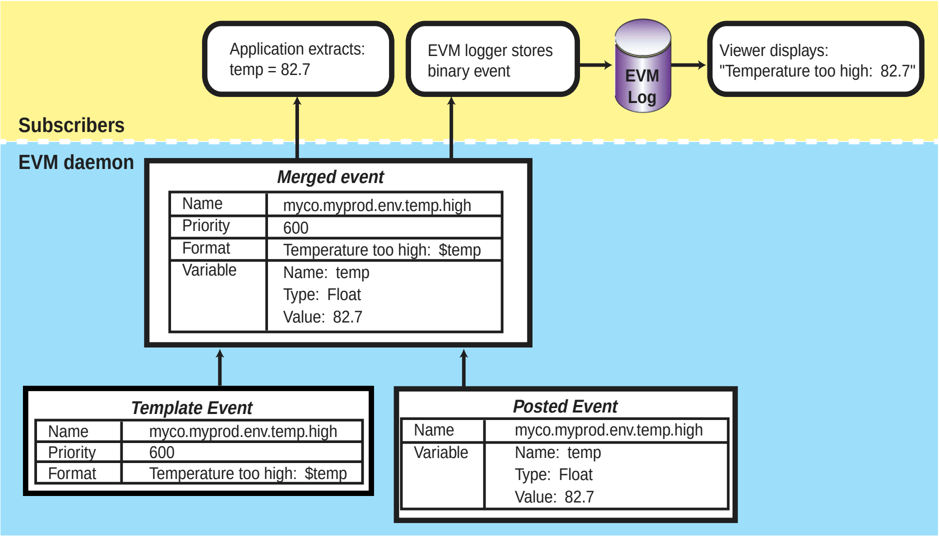
Posted Event Name	Template Event Name	Match
`myco.myprod.env`	`myco.myprod.env`	Yes
`myco.myprod.env.temp.high.70`	`myco.myprod.env.temp`	Yes
`myco.myotherprod`	`myco.myother-prod.start`	No — the posted event has too few components

14.6.3.3 Merging Data Items from Templates and Posted Events

After the EVM daemon has successfully validated a posted event, it merges the data items contained in the posted event with any data items contained in the template, and then distributes the merged event to any clients that have subscribed to the event. The merge process gives the event designer maximum flexibility in deciding which text and data items are to be provided by the template and which are to be provided by the poster.

Figure 14–2 illustrates the concept of event merging.

Figure 14–2: Posted Event and Template Merging



If the same data item is supplied in both the template and the posted event, the posted value is used in the merged event.

The merge process yields a binary event structure that contains a combination of the data items from the posted event and the template. The merged event is distributed to subscribers in its binary form, not as a formatted message, and subscribers must use the EVM API functions to format the event for display or to extract information from it. The API functions are described in Section 14.7.

14.6.3.4 Installing Template Files — Location, Naming, Ownership, and Permission Requirements

Event template files should normally be installed as part of the product or application installation process. System templates are stored in subdirectories under `/usr/share/evm/templates`, while third-party product and local application templates should be stored under the local template directory, `/var/evm/adm/templates`. To enable the EVM daemon to locate the additional templates, a link is provided from the system template directory to the local directory.

To add templates for a product or local application, the application's installation process should create an appropriately named subdirectory of the local template directory and install its templates in the new directory. EVM's template search policy follows symbolic links, so you may also install the templates in a directory more closely associated with the application, and connect it to the local directory with a link.

Template files must have the suffix `.evt`, must be owned by `root` or `bin`, and must have one of the following access permissions: 0600, 0400, 0640, or 0440. Also be sure to give the new directory appropriate permissions.

After you install the file, run (as root) the command `evmreload -d` to cause EVM to recognize the new templates, and then check for any errors. See `evmreload`(8) for details.

14.6.3.5 Checking Event Template Registration

You can determine whether your templates are registered by using `evmwatch` with the `-i` option. For example, the following command lists the names of all event templates registered for the `myapp` application:

```
evmwatch -i -f "[name myco.myprod.myapp]" | evmshow -t "@name"
```

Use the `-d` option of `evmshow` to display the template details. Note that `evmwatch` will not return the templates of any events for which you do not have access authorization, so you may need to be logged in as root to see your templates.

14.6.4 Establishing Translations for Event Text (I18N)

The objective of event internationalization (I18N) is to allow an event's format data item, and the values of any string variables contained in the event, to be converted to another language for display. If you are developing a product that will be used internationally, you can include support for translation of any or all of these items.

Because different users may want to view the same stored event in different languages, language interpretation must be performed on-the-fly when the event is formatted for presentation, not when it is posted. It is impractical for an event to carry the text in all possible languages, so its associated message catalog must be available when the event is formatted. Product developers are responsible for providing message catalogs and for including them for installation along with their products. To handle the case in which the catalog is not available, an internationalized event can carry a default native-language string.

The following data items support internationalization of events:

- I18N message catalog name
- I18N message set identifier (optional)
- I18N message identifier for the format data item
- Separate I18N message identifier for each internationalized string variable
- Default native-language strings for any or all of the previous items in this list

All message identifiers for the event must relate to the same message catalog, and they must all belong to the same message set (1, by default).

In general, the catalog ID, set ID, and message ID for the event format string should all be supplied in the event template because the format string is usually fixed. Where events contain string-type variables, the variables are likely to refer to items such as device names or application names, which usually will not need to be translated, regardless of the language for display — and hence in most cases it will not be necessary to supply a message ID. In the rare cases in which the value of a string variable does need to be translated, the poster must supply the message ID.

Table 14–5 contains some examples of internationalized values for an example event.

Table 14–5: Example Data Item Values for an Internationalized Event

Event Data Item	Value	Message ID
Name	`acme.prod.env.temp`	n/a
Message catalog	`acme_prod.cat`	n/a
Format string	`Temperature of sensor $sensor is $temperature`	541

Table 14–5: Example Data Item Values for an Internationalized Event (cont.)

Event Data Item	Value	Message ID
String variable “sensor”	`S27`	n/a
String variable “temperature”	`high`	542

English and French versions of `temperature.cat` contain:

- English
  - `541:Temperature of sensor $sensor is $temperature`
  - `542:high`
  - `543:low`
- French
  - `541:La temperature du senseur $sensor est $temperature`
  - `542:haute`
  - `543:basse`

When the viewer has to display the event, it invokes a format function, which, depending on the user’s locale setting, returns either of these strings *if* it can find the appropriate catalog file and *if* the file contains the specified messages:

```
"Temperature at sensor S27 is high"
"La temperature du senseur S27 est haute"
```

If the format function cannot interpret the events from a catalog, it uses the values that are carried in the event and returns the following message regardless of the user’s locale:

```
"Temperature at sensor S27 is high"
```

If an event file is passed to another system for analysis, it is possible that the associated catalog file will be unavailable. If the event includes default values, as in the previous example, it can still be displayed in the original language. If the event does not include defaults, the formatted string can still show the event name, along with a dump of any variables. For example:

```
Event "myco.myprod.env.temp": $sensor = "S27" $temperature = "high"
```

See the *Writing Software for the International Market* manual for additional information on I18N issues.

14.7 The EVM Programming Interface

EVM events are opaque data structures that are accessed and manipulated by using EVM’s application programming interface (API) functions.

A Java Native Interface (JNI) package is provided to support Java programs running on a Tru64 UNIX system. For information about the Java interface, use a web browser to view the following file:

```
file:/usr/share/doclib/evm/java_api/help-doc.html
```

The following sections provide programming information and examples of some of the operations commonly performed on EVM events.

14.7.1 EVM Header File

Programs that use EVM functions must include the following header file:

```
#include <evm/evm.h>
```

14.7.2 EVM API Library

Programs that use EVM API functions must link against either the shared library `libevm.so` or the static library `libevm.a`. The shared library is located in the root-partition directory, `/shlib`, and consequently is available to programs that may need to be run when the system is in single-user mode. However, because the EVM daemon is not started until run level 2, any attempt to connect to the daemon while in single-user mode will fail. There is a symbolic link from `/usr/shlib` to the shared library in `/shlib`, making the library available to programs linked using the default library search path.

14.7.3 Return Status Codes

The status codes returned by EVM functions are enumerated in the header file `evm/evm.h`. EVM functions commonly return the following values:

- `EvmERROR_NONE` — The operation completed without error.
- `EvmERROR_INVALID_ARGUMENT` — One of the arguments passed to a function was invalid.
- `EvmERROR_INVALID_VALUE` — A structure contained an invalid value.
- `EvmERROR_NO_MEMORY` — An operation failed because an attempt to acquire heap memory failed.
- `EvmERROR_NOT_PRESENT` — A requested item is not present in the event.

Refer to the `evm/evm.h` file for a complete list of status codes.

14.7.4 Signal Handling

The EVM API does not use signals in its normal processing, and in general does not interfere with an application program's use of signals. However, because the Tru64 UNIX system's default action is to silently terminate a process that attempts to write to a local connection if no process exists to read the data, there is a danger that a client process could exit without trace if the EVM daemon terminated before or during activity on the connection.

To prevent this from happening, the `EvmConnCreate( )` function checks whether the caller has already established a handler for SIGPIPE, and, if not, the function installs a default handler. The handler takes no action if the signal occurs, but its presence prevents the client from terminating. A program can override the EVM handler by setting its own handler either before or after the call to `EvmConnCreate( )`. If it is important that the program takes the default action, set the action to SIG_DFL after calling `EvmConnCreate( )`.

See `signal`(2) for more information.

14.7.5 EVM in Multithreaded Programs

All EVM API functions are thread-safe; in the few cases where it is necessary to use internal static storage, they use locks to protect the storage from simultaneous access by separate threads. Nevertheless, if you are using EVM API calls in a multithreaded program, you need to take certain precautions to avoid synchronization errors:

1. If possible, restrict the use of any entity returned by any API function to the thread in which it was established. These entities include:

Entity	**Type**	**Returned by**
Connection context	`EvmConnection_t`	`EvmConnCreate()`

Event	`EvmEvent_t`	`EvmEventCreate()`
		`EvmEventCreateVa()`
		`EvmEventRead()`
		`EvmEventDup()`

Data item	`EvmItemValue_t`	`EvmItemGet()`
Data item list	`EvmItemList_t`	`EvmItemListGet()`
Variable	`EvmVarValue_t`	`EvmVarGet()`
Variable list	`EvmVarList_t`	`EvmVarListGet()`
Event filter	`EvmFilter_t`	`EvmFilterCreate()`
Connection fd	`EvmFd_t`	`EvmConnFdGet()`

2. If it is necessary to refer to these entities in more than one thread, it is essential that you protect them against simultaneous access or update by using locks.

If you do not follow these rules, it is highly likely that random errors will occur.

14.7.6 Reassigning and Replicating EVM Events

If you need to reassign EVM events after they have been created, received, or read, it may be useful to understand how an event is held in memory.

The `EvmEvent_t` type defines a pointer to a short handle structure that holds some control information and a pointer to the event body. When you use `EvmEventCreate( )` or any related function to create a new event, the function allocates heap memory separately for the handle and the event body, stores the address of the body in the handle, and returns the address of the handle as the function's `EvmEvent_t` output argument. You must use the returned pointer any time you need to refer to the event.

If you modify the event (by adding a variable, for example), it is likely that the space used by the event body will be freed and reallocated in a new location. When this happens, the address stored in the handle is automatically updated to reflect the new location; hence, the reallocation is completely transparent to your program. The location of the handle remains unchanged for the life of the event.

If you need to transfer an event from one variable to another, you can do so using a simple C-language assignment statement between variables of type `EvmEvent_t`. Because the value you hold is simply a pointer to a constant location — the event's handle — you can, if you wish, continue to use both of the variables to refer to the event. However, if you subsequently use `EvmEventDestroy( )` to destroy the event, you must be sure to discard both references.

You must also be aware that reassigning an event copies just the reference to its handle; it does not copy the event body. If you need to make a completely independent copy of the event, you can do so using the `EvmEventDup( )` function call.

14.7.7 Callback Functions

EVM posting and subscribing clients connect to the EVM daemon using the `EvmConnCreate( )` function call, and must specify one of three possible response modes: `EvmRESPONSE_IGNORE`, `EvmRESPONSE_WAIT` or `EvmRESPONSE_CALLBACK`. The modes are described in `EvmConnCreate`(3).

Subscribing clients are required to specify `EvmRESPONSE_CALLBACK` as the response mode, and incoming events are passed to them by the callback function that you supply as the fourth argument to `EvmConnCreate( )`. An example of the use of a callback function by a subscribing client is given in Section 14.7.12.6.

When working with a callback function, it is important to understand that your function is not called asynchronously, in the manner of a signal handler. Rather, your program must monitor the connection for input activity, using `EvmConnWait( )` or `select( )` or a related function, and then call `EvmConnDispatch( )` to handle the activity. `EvmConnDispatch( )` then reads an incoming message from the connection and invokes your callback function, if necessary. The possible reasons for the callback being invoked are listed in `EvmCallback`(5).

As with any function, the arguments passed to your callback function are passed on the program stack and are available only within the scope of the function; therefore, if you want to save any values for use after you have returned from the callback, you must copy them to global memory space before returning.

If the callback is reporting an incoming event and you want to preserve the event instead of handling it and then destroying it within the callback, you can simply declare a globally accessible variable of type `EvmEvent_t` and assign the incoming event to it, for example:

```
/* In global declarations */
EvmEvent_t SavedEvent;
...

/* In your callback function */
SavedEvent = cbdata->event;
```

In this case, you must not destroy the event in the callback function, because the assignment copies just the reference to the event, not the event body. You must still arrange to destroy the event from elsewhere after you have finished with it; otherwise, you may have introduced a memory leak.

See Section 14.7.6 for a description of event assignment.

14.7.8 Choosing a Connection Policy

If your program is a posting client, you need to choose a policy for connecting to the EVM daemon. Choose from the following options:

- Create a connection when the program starts, and maintain it until the program terminates (*permanent* connection).
- Create a connection each time that you need to post an event, and destroy it immediately afterwards (*transient* connection).

The cost in processing time of creating an EVM connection is significant because the setup and authentication protocol requires several message transactions as well as some file I/O. As a general rule, maintain a permanent connection if you expect to post events reasonably frequently during normal program operation. If you choose this option, your code must be prepared to deal with unexpected disconnection in case the EVM daemon is terminated. See Section 14.7.9 for information about handling disconnection.

If you expect to post events infrequently — for example, only if unexpected errors occur — consider using a transient connection. Although a transient connection is more expensive in set-up time if multiple events are posted, it eliminates both the system resource cost of a permanent connection and the need for your code to deal with an unexpected disconnection. The simplest way to post events to the local daemon with a transient connection is to pass NULL as the connection argument to `EvmEventPost( )` or `EvmEventPostVa( )`.

If you do not require a permanent connection, but you expect to post several events over a short period under some circumstances, consider creating a temporary connection that you will destroy when the activity has been completed, rather than posting successive events using a separate connection for each.

Subscribing clients must maintain permanent connections to be assured of receiving events.

14.7.9 Handling Disconnections

If your program is a **subscribing client**, it is particularly important that you handle disconnection from the EVM daemon correctly. A disconnect should not occur under normal operation, but may occur in a system that is being tested or if a fault is encountered. If the EVM daemon terminates, it will usually be restarted automatically within a few seconds by the Essential Services Monitor daemon. See `esmd`(8) for more information.

Although you should always test the return code from functions dealing with a connection, it is especially important to do so in a subscribing client, which may spend most of its time waiting for activity on a connection.

If a disconnect occurs, your program will drop out of its `select( )` or `EvmConnWait( )` call, and subsequent calls to `EvmConnCheck( )` and `EvmConnDispatch( )` will return failure status codes. When this happens, you should not return immediately to the `select( )` or `EvmConnWait( )` call because that would result in a CPU-bound loop. Instead, determine whether the status code indicates a connection error and, if so, destroy the connection using `EvmConnDestroy( )` and then attempt to reconnect. If the initial attempt to reconnect fails, you should arrange to try again after a reasonable delay and continue to retry periodically until the connection is re-established. Because the daemon is usually restarted automatically following any failure, the recommended retry interval is once per second for the first 60 seconds, and every five seconds thereafter, until the connection is restored.

Note that not *all* error codes returned by the connection functions indicate a disconnect. In particular, it is very possible that the program will drop out of `EvmConnWait( )` as a result of a signal. If this is indicated, the correct action is to return immediately to the `EvmConnWait( )` call.

If your program is a **posting client**, you will usually need to deal with a disconnection only if you are using a permanent connection (see Section 14.7.8). You must be sure to examine the return status from `EvmEventPost( )` or `EvmEventPostVa( )`, and, if necessary, re-establish the connection and retry the post operation.

14.7.10 Missed Events

In a system with a high level of event activity, it is possible for a client that has subscribed for many of the events being posted to fail to receive some of them. This is particularly likely if the client takes a long time to process the events — for example because it has to write each event to disk. An example of such a client is the EVM logger, which recognizes when it has missed events and reports the failure by logging a special "missed events" event. If there is a danger that your application might miss events, you should ensure that the system is configured to minimize the risk, and write your programs to react appropriately.

When the EVM daemon sends an event to a subscribing client, it does so with a connection that depends upon fixed-size communication (send and receive) buffers. If the client does not deal with its incoming events within a reasonable time and there is a heavy event load, the buffers may fill and the daemon will be unable to send further events. The default receive buffer size is set to the value of the system-wide socket buffer maximum, as defined by the sysconfig parameter `sb_max`. The send buffer size is 32,767, which is the system default for socket buffers.

Because the EVM daemon is a critical resource for many system components and applications, it cannot block while waiting for a client to clear its buffer. Consequently, if it fails to write to a connection because the buffer is full, the daemon marks the connection as blocked and continues with its other activities. When the client eventually reads its input and frees space in the buffer, the EVM daemon completes the failed write and the client will receive the event. However, if in the meantime any other events arrive that should be sent to the blocked subscriber, the daemon does not attempt to send them; instead, it counts how many have been missed and reports the number to the subscriber when the connection becomes unblocked. The subscriber must take any appropriate action, but there is no way for it to know which events have been missed.

As a result of an increase in the default receive buffer size, the likelihood of missing events is lower than it was in earlier releases. However, it is still possible for your application to miss incoming events under extreme circumstances. The likelihood of this happening depends on several factors.

- The speed and load of the system
- The set of subscribed-for events
- The frequency with which events are being posted
- The size of the connection’s input buffer
- The sizes of the events being posted

To minimize the risk of missing incoming events, design your application to deal with incoming events as quickly and efficiently as possible and to take appropriate action when it is notified that it has missed events. See Section 14.7.12.10 for an example of a program that deals with missed events.

If your application consistently misses events, you can increase the receive buffer size by changing a system parameter. The receive buffer size is set to the default system socket buffer maximum. To determine the current size of this parameter, enter the following command:

```
sysconfig -q socket sb_max
```

To change the run-time value of this parameter, which would remain in effect until the next reboot, enter the following command.

```
sysconfig -r socket sb_max=new_value
```

This change will only affect new EVM connections.

If you want to make the change permanent, use `sysconfigdb` or `dxkerneltuner` to make the change. Refer to `sysconfigdb`(8) or `dxkerneltuner`(8) for more information.

14.7.11 Using Event Filters

An event filter is used to identify the set of events in which you are interested. After it is established, a filter evaluator is given a string defining the events of interest, and can then be passed a series of events, returning for each a Boolean indication of whether the event passes the filter.

Filters are used in EVM subscribing client programs to specify the set of events for which they want to subscribe, and may also be used in determining the action to be taken for the events they receive. To limit the set of events returned, filters can also be used by several command-line utilities as start options. See the *System Administration* manual for information on the use of filters by command-line utilities, and see `EvmFilter`(5) for a formal description of filter syntax.

See Section 14.7.12.8 for an example of how to use filters.

14.7.12 Sample EVM Programming Operations

Examples of the following operations are provided in the sections that follow:

- Performing simple event manipulations (Section 14.7.12.1)
- Using variable-length argument lists (Section 14.7.12.2)
- Adding and retrieving variables (Section 14.7.12.3)
- Posting events (Section 14.7.12.4)
- Reading and writing events (Section 14.7.12.5)
- Subscribing for event notification (Section 14.7.12.6)
- Handling multiple I/O sources (Section 14.7.12.7)
- Using filter evaluators (Section 14.7.12.8)
- Matching event names (Section 14.7.12.9)
- Dealing with missed events (Section 14.7.12.10)

14.7.12.1 Performing Simple Event Manipulations

All EVM clients need to work with the EVM event, an opaque binary structure that can hold standard data items and variables. Example 14–2 shows you how to create an event, add items to it, and then retrieve the items from it.

The example introduces the following functions:

- `EvmEventCreate` — Creates an empty event. (See `EvmEventCreate`(3) for details.)
- `EvmEventDestroy` — Destroys a previously created event, freeing its memory. This function must be used if it is necessary to free an event. Although the event reference points to a structure allocated from the heap, that structure contains references to other structures, and hence using `free( )` directly on the event reference will result in lost memory. (See `EvmEventDestroy`(3) for details.)
- `EvmItemSet` — Sets data item values in the event. The list of items and variables to be supplied is the same as that supplied for `EvmEventCreateVa`. (See `EvmItemSet`(3) for details.)
- `EvmItemGet` — Supplies the value of a specified event data item. (See `EvmItemGet`(3) for details.)
- `EvmItemRelease` — Releases any memory that was allocated when a specified data item was retrieved from an event by `EvmItemGet`(). (See `EvmItemRelease`(3) for details.)

Example 14–2: Performing Simple Event Manipulations

```
#include <stdio.h>
#include <evm/evm.h>

main()
{
    EvmEvent_t          event;
    EvmItemValue_t      itemval;
    EvmStatus_t         status;

    EvmEventCreate(&event); 1

    EvmItemSet(event,EvmITEM_NAME,"myco.examples.app.started");  2
    EvmItemSet(event,EvmITEM_PRIORITY,200);

    status = EvmItemGet(event,EvmITEM_NAME,&itemval);  3
    if (status == EvmERROR_NONE)
    {   fprintf(stdout,"Event name: %s\n",itemval.NAME);
        EvmItemRelease(EvmITEM_NAME,itemval);
    }

    EvmEventDestroy(event);  4
}
```

1 You can create an empty event with `EvmEventCreate( )`. When you use this function, you supply a pointer to the event handle, and you receive an event that contains no standard data items. Even though it is empty, the event does take up memory, and you must eventually use `EvmEventDestroy( )` to free the space.

2 You can add any of the standard data items to the event using `EvmItemSet( )`. In most cases, however, the only item you will want to add in your program is the name of the event — other standard items will be automatically added when you post the event, or are better included in the event template. See `EvmItemSet`(3) for a list of the settable items.

3 You can retrieve any item from the event using `EvmItemGet( )`. The value is copied from the event into storage referenced through your `EvmItemValue_t` structure, so you must use `EvmItemRelease( )` to release the storage when you have finished with it. Retrieving the item does not remove it from the event; you receive a copy and you can get it as many times as you want.

This piece of code retrieves the event's name (which was added earlier), prints its value, and then releases the storage. You should always examine the return status from the get operation because you may be requesting an item that is not present in the event.

4 When you have finished with the event, free the storage space used by the event.

14.7.12.2 Using Variable-Length Argument Lists

You can reduce the size and improve the efficiency of your code by creating an event and adding items to it in a single step, using the varargs (variable-length argument list) version of the create function — and you can add items to an existing event efficiently by using the varargs version of the item-set function.

Example 14–3 introduces the following functions:

- `EvmEventCreateVa` — Creates an event and supplies item names and values in a single call. (See `EvmEventCreateVa`(3) for details.)
- `EvmItemSetVa` — Sets data item values in the event. The list of items and variables to be supplied is the same as that supplied for `EvmEventCreateVa`. (See `EvmItemSetVa`(3) for details.)

Example 14–3: Using Variable-Length Argument Lists

```
#include <stdio.h>
#include <evm/evm.h>

main()
{
    EvmEvent_t          event;

    EvmEventCreateVa(&event, [1]
                    EvmITEM_NAME,"myco.examples.app.started",
                    EvmITEM_PRIORITY,200,
                    EvmITEM_NONE);

    EvmItemSetVa(event,
                EvmITEM_NAME,"myco.examples.app.finished", [2]
                EvmITEM_PRIORITY,100,
                EvmITEM_NONE);

    EvmItemSetVa(event,
                EvmITEM_VAR_UINT16,"exit_code",17, [3]
                EvmITEM_VAR_STRING,"progname","my_app",
                EvmITEM_NONE);

    EvmEventDump(event,stdout); [4]

    EvmEventDestroy(event); [5]
}
```

[1] Each item you include in `EvmEventCreateVa( )` must have an identifier and a value, and the argument list must be terminated with an `EvmITEM_NONE` identifier.

[2] The varargs version of `EvmItemSet( )` uses the same style of argument list as `EvmEventCreateVa( )`. In this example, items that are already present in the event are being added, so the new values just replace the old ones.

3 You can easily include variable data item in a varargs list by using the EvmITEM_VAR_*Xxx* item identifier, where *Xxx* is the type of the variable. If you include a variable in this way, it is very important that you follow the identifier with the correct number of arguments to describe the variable. You must always supply two further arguments: a string containing the name of the variable, and a value argument. The type of the value argument depends on the type of the variable — in this example, the first variable requires a string and the second requires an integer. Certain types of variable require an additional argument — see EvmItemSet(3) for more details.

4 The call to EvmEventDump( ) displays a formatted dump of the event on stdout, to demonstrate that the expected data items and variables have been added correctly.

5 When you have finished with the event, free the storage space that it uses.

14.7.12.3 Adding and Retrieving Variables

In the previous example, variable items were added to an event by including them in a varargs list, using EvmItemSetVa( ). You can add or change variable values by using any of the varargs functions that take item identifiers as arguments.

Example 14–4 shows another way to add variable data values to an existing event and also shows how to retrieve the values of the variables. The EVM library includes primitive get and set functions that use a data structure to describe a variable, as well as a set of convenience functions that simplify programming by encapsulating the handling of the structure. The example illustrates the use of both types of function.

The example introduces the following functions:

- EvmVarSet — Sets the value of a named variable data item in an event. This function is used both for adding a variable and for altering the value of an existing variable. (See EvmVarSet(3) for details.)
- EvmVarGet — Returns details of a specified event variable in a given variable structure. The caller must free any memory used by the variable by calling EvmVarRelease(). (See EvmVarGet(3) for details.)
- EvmVarRelease — Releases any memory that was allocated when the specified variable was retrieved from an event by EvmVarGet(). It does not release the specified variable structure, only any heap storage referenced by the structure. (See EvmVarRelease(3) for details.)

Example 14–4: Adding and Retrieving Variables

```
#include <stdio.h>
#include <evm/evm.h>
void  main()
{
   EvmEvent_t          event;
   EvmStatus_t         status;
   EvmVarValue_t       varval_1, varval_2; 1
   EvmVarStruct_t      varinfo;
   EvmString_t         progname;
   EvmUint16_t         exit_code;

   EvmEventCreateVa(&event,
           EvmITEM_NAME,"myco.examples.app.finished",
           EvmITEM_NONE);
   /*
    * Set and retrieve some values using basic set/get functions:
    */
   varval_1.STRING = "my_app";
   varval_2.UINT16 = 17;

   EvmVarSet(event,"progname",EvmTYPE_STRING,varval_1,0,0); 2
   EvmVarSet(event,"exit_code",EvmTYPE_UINT16,varval_2,0,0);

   status = EvmVarGet(event,"progname",&varinfo); 3
   if (status == EvmERROR_NONE)
   {   fprintf(stdout,"Program name: %s\n",varinfo.value.STRING);
       EvmVarRelease(&varinfo);
   }

   status = EvmVarGet(event,"exit_code",&varinfo);
   if (status == EvmERROR_NONE)
   {   fprintf(stdout,"Exit code: %d\n",varinfo.value.UINT16);
       EvmVarRelease(&varinfo);
   }

   /*
    * Set and retrieve the same values using convenience functions:
    */
   EvmVarSetString(event,"progname","my_app"); 4
   EvmVarSetUint16(event,"exit_code",17);

   status = EvmVarGetString(event,"progname",&progname,NULL); 5
   if (status == EvmERROR_NONE)
       fprintf(stdout,"Program name: %s\n",progname);
   free(progname); 6

   status = EvmVarGetUint16(event,"exit_code",&exit_code);
   if (status == EvmERROR_NONE)
       fprintf(stdout,"Exit code: %d\n",exit_code);

   EvmEventDestroy(event); 7
}
```

1 When using a primitive function to add a variable to the event, you must first place the value in a union of type `EvmVarValue_t`. The members of the union have the same names as the EVM variable types.

2 Use `EvmVarSet( )` to add the variables to the event, giving them meaningful names. The final two arguments are set to 0 unless you

are adding an opaque variable or supplying an I18N message ID for a string variable.

3 You can retrieve the value of any variable by passing its name to EvmVarGet( ), which copies the value into an EvmVarStruct_t structure. The structure also contains the name, type, and size of the variable, so you can write generic code to handle any type of variable. Retrieving a variable does not remove the variable from the event, and you can retrieve it as many times as you want. The returned information always take up space in heap memory, so you must use EvmVarRelease( ) to clean up after you have finished with the value.

4 The EvmVarSetString( ) and EvmVarSetUint16( ) functions, along with other functions in the EvmVarSetXxx( ) family, are simpler alternatives to the EvmVarSet( ) function. They require fewer arguments, and there is no need to set up a value structure.

If you want to supply an I18N message-id with a string variable, use the EvmVarSetStringI18N( ) function.

5 The EvmVarGetString( ) and EvmVarGetUint16( ) functions, along with other functions in the EvmVarGetXxx( ) family, are simpler alternatives to the EvmVarGet( ) function. They require fewer arguments, and there is no need to set up a variable information structure or to follow the retrieval operation with a call to EvmVarRelease( ).

6 If you use a convenience function to retrieve a string or opaque variable, the function allocates heap space for the value and returns a pointer to the heap. You must use free( ) to release the heap space after you have finished with the value.

7 After you have finished with the event, free the storage space used by the event.

14.7.12.4 Posting Events

The most likely reason for creating an event is to post it. Posting an event results in the event being distributed to subscribers by the EVM daemon. Before you can post the event, you must create a posting connection to the daemon.

Example 14–5 shows how to create the connection, post the event, and disconnect.

The example introduces the following functions:

- EvmConnCreate — Establishes a connection between an application program and EVM, and defines how I/O activity is to be handled. A separate connection must be established for each type of connection:

posting, listening (subscribing), or service. (See EvmConnection(5) and EvmConnCreate(3) for details.)

- EvmEventPost — Posts an event to EVM. (See EvmEventPost(3) for details.)
- EvmConnDestroy — Destroys a specified connection. (See EvmConnDestroy(3) for details.)

You can use the simpler EvmConnCreatePoster( ) macro in place of EvmConnCreate( ) to create a posting connection. This macro requires fewer arguments than EvmConnCreate( ), and although it also offers fewer connection options, it is suitable for use by most applications that need to post events to the local system.

If your program uses transient EVM connections, you can omit calls to EvmConnCreate( ) or EvmConnDestroy( ) and pass NULL as the connection argument to EvmEventPost( ). Connections use significant processing time to establish, so you should use only transient connections if they are appropriate for your program. See Section 14.7.8 and EvmEventPost(3) for more information.

You can create, post, and destroy an event in a single call using the EvmEventPostVa( ) function. See EvmEventPost(3) for more information.

Example 14–5: Posting Events

```
#include <stdio.h>
#include <evm/evm.h>

void main()
{   EvmEvent_t          event;
    EvmStatus_t         status;
    EvmConnection_t     conn;

    status = EvmConnCreate(EvmCONNECTION_POST, EvmRESPONSE_WAIT, 1
                        NULL,NULL,NULL,&conn);
    if (status != EvmERROR_NONE)
    {   fprintf(stderr,"Failed to create EVM posting connection\n");
        exit(1);
    }

    EvmEventCreateVa(&event, 2
            EvmITEM_NAME,"myco.examples.app.error_detected",
            EvmITEM_NONE);

    status = EvmEventPost(conn,event);
    if (status != EvmERROR_NONE)
    {   fprintf(stderr,"Failed to post event\n");
        exit(1);
    }

    EvmEventDestroy(event); 3
    EvmConnDestroy(conn);
}
```

1 You can create a connection to the EVM daemon using `EvmConnCreate( )`. The connection remains in place until your program exits, or until you explicitly destroy it with `EvmConnDestroy( )`. The first two arguments to `EvmConnCreate( )` specify that the connection will be used for posting events, and that you want the post function to wait for the daemon to acknowledge acceptance of the event before returning, so that you can take action if the attempt fails. (See `EvmConnCreate`(3) for other response options.) The `NULL` value for the third argument indicates that you are making a connection to the EVM daemon running on the local system — you will almost always specify `NULL` here; see `EvmConnCreate`(3) for details about remote connection. The fourth and fifth values are used for other response types, and should always be `NULL` for wait-mode responses. The final argument receives the handle to the connection — you must supply the returned value in all future calls being made on this connection.

2 Create an event and post it.

3 Clean up by destroying the event and the connection. If you expect to be posting events periodically, it may be better not to destroy the connection, but to reuse it for all future events. This will save you

the overhead of re-establishing the connection each time you have something to post.

14.7.12.5 Reading and Writing Events

You will need to use the EVM read and write functions if you are writing a program that performs any of the following operations:

- Stores events in a file
- Passes events to another process through a pipe or socket connection
- Analyzes events stored in a file
- Receives events from a process other than the EVM daemon

You cannot write events directly using the standard UNIX write functions because the event handle only contains a pointer to the body of the event — and because the location of the body may change each time the event is modified. Conversely, when you read an event, it is not enough just to read the body; a handle has to be created to allow you to reference the event through the API functions.

Example 14–6 shows you how to write events to a file, to read them from a file into your program, and to validate them.

The example introduces the following functions:

- `EvmEventWrite` — Writes an event to an open file descriptor. (See `EvmEventWrite`(3) for details.)
- `EvmEventRead` — Creates a new event structure and populates it with an event read from a file descriptor. `EvmEventDestroy( )` must be used to free the new event. (See `EvmEventRead`(3) for details.)
- `EvmEventValidate` — Performs a data integrity check on the event. This check is intended to validate an event that has just been received over a connection or retrieved from storage. (See `EvmEventValidate`(3) for details.)

Example 14–6: Reading and Writing Events

```
#include <stdio.h>
#include <fcntl.h>
#include <evm/evm.h>

void  main()
  {   EvmEvent_t          event_in,event_out;
      EvmStatus_t         status;
      EvmItemValue_t      itemval;
      int                 fd;

      EvmEventCreateVa(&event_out, 1
              EvmITEM_NAME,"myco.examples.app.saved_event",
              EvmITEM_NONE);
      fd = open("eventlog",O_RDWR | O_CREAT | O_TRUNC, 2
```

Example 14–6: Reading and Writing Events (cont.)

```
                          S_IRUSR | S_IWUSR);
      if (fd < 0)
      {   fprintf(stderr,"Failed to open output log file\n");
          exit(1);
      }
      status = EvmEventWrite(fd,event_out);
      if (status != EvmERROR_NONE)
      {   fprintf(stderr,"Failed to write event to log file\n");
          exit(1);
      }
      lseek(fd,0,SEEK_SET); 3
      status = EvmEventRead(fd,&event_in);
      if (status != EvmERROR_NONE)
      {   fprintf(stderr,"Failed to read event from log file\n");
          exit(1);
      }
      status = EvmEventValidate(event_in); 4
      if (status != EvmERROR_NONE)
      {   fprintf(stderr,"Event read from logfile is invalid");
          exit(1);
      }
      status = EvmItemGet(event_in,EvmITEM_NAME,&*itemval); 5
      if(status == EvmERROR_NONE)
      {   fprintf(stdout,"Event name: %s\n",itemval.NAME);
          EvmItemRelease(EvmITEM_NAME,itemval);
      }
      EvmEventDestroy(event_in); 6
      EvmEventDestroy(event_out);
}
```

1 Create an event containing a name.

2 Create an output log file and use `EvmEventWrite( )` to write the event to it. You can write the event to any file descriptor, including a pipe to another process — but because the event is a binary data package, do not write it directly to a terminal or printer.

3 Read the event back in using `EvmEventRead( )`. Note that you produce a different event this time, and you have to supply a pointer to the event handle, not the handle itself.

4 Because the incoming event has been outside the control of this process, it is important to verify its integrity. Use the `EvmEventValidate( )` function to do this each time you either read an event from a file or receive it from any process other than the EVM daemon.

5 You can show that the event just read is the same event that was just written out by retrieving and displaying its name.

6 Free the space used by the event.

14.7.12.6 Subscribing for Event Notification

A program that subscribes for receiving event notifications must perform the following operations:

- Create a listening connection to the EVM daemon
- Tell the daemon which events it is interested in by passing a filter string
- Monitor event activity on the connection and be prepared to handle each event as it arrives

Example 14–7 waits for events to arrive and displays each incoming event on `stdout`.

The example introduces the following functions:

- `EvmConnSubscribe` — Requests notification of any posted events that match the supplied filter. (See `EvmConnSubscribe`(3) for details.)
- `EvmConnWait` — Blocks until activity is detected on a specified connection. When activity is detected, the calling program can call `EvmConnDispatch( )` to handle the activity. (See `EvmConnWait`(3) for details.)
- `EvmConnDispatch` — Handles any outstanding I/O on a specified connection by calling the program's callback function as needed. (See `EvmConnDispatch`(3) for details.)
- `EvmEventFormat` — Formats an event. (See `EvmEventFormat`(3) for details.)

Note

You may prefer to use the simpler `EvmConnCreateSubscriber( )` macro in place of `EvmConnCreate( )` to create a subscribing connection. This macro requires fewer arguments than `EvmConnCreate( )`, and although it also offers fewer connection options, it is suitable for use by most subscribing applications. See `EvmConnCreate`(3) for more information.

Example 14–7: Subscribing for Event Notification

```
#include <stdio.h>
#include <evm/evm.h>

void EventCB(EvmConnection_t conn, EvmCallbackArg_t cbarg,
           EvmCallbackData_t *cbdata);

/*===============================================
 * Function: main()
 *===============================================*/
main()
```

Example 14–7: Subscribing for Event Notification (cont.)

```
{   EvmConnection_t     conn;
    EvmStatus_t         status;

    status = EvmConnCreate(EvmCONNECTION_LISTEN, EvmRESPONSE_CALLBACK,
                        NULL,EventCB,NULL,&conn); 1
    if (status != EvmERROR_NONE)
    {   fprintf(stderr,"Failed to create EVM listening connection\n");
        exit(1);
    }
    status = EvmConnSubscribe(conn,NULL,"[name *.evm.msg.user]"); 2
    if (status != EvmERROR_NONE)
    {   fprintf(stderr,"Failed to subscribe for event notification\n");
        exit(1);
    }
    for (;;) 3
    {   status = EvmConnWait(conn,NULL);
        if (status == EvmERROR_NONE)
        {   fprintf(stderr,"Connection error\n");
            exit(1);
        }
        if (EvmConnDispatch(conn) != EvmERROR_NONE)
        {   fprintf(stderr,"Connection dispatch error\n");
            exit(1);
        }
    }
}
/*====================================================
 * Function: EventCB()
 *===================================================*/
void EventCB(EvmConnection_t conn, EvmCallbackArg_t cbarg, 4
                EvmCallbackData_t *cbdata)
{   char buff[256];

    switch (cbdata->reason) { 5
    case    EvmREASON_EVENT_DELIVERED:
            EvmEventFormat(buff, sizeof(buff), cbdata->event);
            fprintf(stdout,"Event: %s\n",buff);
            EvmEventDestroy(cbdata->event); 6
            break;
    default: 7
            break;
    }
}
```

1 Use EvmConnCreate( ) to establish a connection to the EVM daemon, this time specifying a listening connection. The following arguments appear in this code segment:

- The first two arguments to EvmConnCreate( ) specify that the connection will be used for listening and that, when events arrive, you want to be notified through a callback function.
- The NULL third argument indicates that you are making a connection to the local EVM daemon; see EvmConnCreate(3) for details about remote connection.

- The fourth argument specifies the callback function to be called when events arrive — EventCB is the next function.
- The fifth argument is the callback argument — a value that will be passed to the callback function each time it is called to service this connection. You can use this argument for anything you want, but in this example it is just set to NULL.
- The final argument receives the handle to the connection.

2 The next step is to use the EvmConnSubscribe( ) function to let the EVM daemon know which events you are interested in receiving. In this case, you will watch for user messages that can be posted with evmpost. These events have the name sys.unix.evm.msg.user. Note that calling EvmConnSubscribe( ) results in your callback function being invoked with a reason code of EvmREASON_SUBSCRIBE_COMPLETE when you call EvmConnDispatch( ).

3 This example does not have anything to do except wait for arriving events, so it loops forever, using EvmConnWait( ) to watch for activity on the connection. Passing NULL as the second argument to EvmConnWait( ) indicates that you do not want to time out if there is no activity. EvmConnWait( ) will return each time *any* activity occurs — not always because an event has arrived. For example, the EVM daemon may have sent some other message or the process may have been interrupted by a signal. Each time the function returns without error, you must call EvmConnDispatch( ) to handle the activity. This will usually (but not always) result in EventCB( ), the callback function, being called.

4 The callback function is called by EvmConnDispatch( ) whenever it reads a message from the daemon that needs to be handled by application code. This includes incoming events; remember, however, that you may need to consider other types of messages in your application. The following arguments appear in this code segment:

- The first argument to the callback function is the connection handle. This is useful to identify the connection in some circumstances, but you may find that you do not have a use for it.
- The second argument is the callback argument that you supplied when you established the connection. You may choose to use this value to identify the connection, or (in C++ code) to pass a pointer to an object instance, or you may not use it at all.
- The final argument is the callback data, a pointer to a structure containing information about the callback. The code must examine the callback data structure to find out why it was called. The structure includes a reason code, a status value, and (if an event is

being delivered) a pointer to the event. See EvmCallback(5) for details about the structure.

For this example, you are looking for incoming event messages, indicated by the EvmREASON_EVENT_DELIVERED reason code. As each event comes in, you format and display it, and then (because you are responsible for the space it consumes) destroy it after you are done with it.

5 The action taken always depends on the reason for the callback. In this case, the reason is event delivered, so you use the EvmEventFormat( ) function to format the event for display, and then you print it to stdout.

Note that EvmEventFormat( ) produces a text line that is formed from the event's format data item (if present), and places the result in the buffer that you supply. Formatting does not cause the event itself to change.

6 The delivered event is using heap space, and it is the responsibility of the application to free the space after it has finished with the event.

7 Because we made a subscription request earlier, the function will also be invoked with a callback reason of EvmREASON_SUBSCRIBE_COMPLETE. For this example, this and any other reason codes can be ignored.

14.7.12.7 Handling Multiple I/O Sources

If you are writing a program that has responsibilities other than just listening for events, you may not want to use EvmEventWait( ) to wait for events to arrive. For example, you may want to use the select system call to wait for I/O activity on multiple file descriptors, including the EVM connection.

Example 14–8 shows how to handle the EVM connection in conjunction with input from stdin.

The example introduces the following functions:

- EvmConnFdGet — Returns the file descriptor (file number) associated with a specified connection. (See EvmConnFdGet(3) for details.)
- EvmConnCheck — Checks whether any I/O activity is outstanding on a specified connection. (See EvmConnCheck(3) for details.)

Example 14–8: Handling Multiple I/O Sources

```
#include <stdio.h>
#include <sys/time.h>
#include <evm/evm.h>

void HandleInput();
void EventCB(EvmConnection_t conn, EvmCallbackArg_t cbarg,
                EvmCallbackData_t *cbdata);
```

Example 14–8: Handling Multiple I/O Sources (cont.)

```
/*================================================
 * Function: main()
 *================================================*/
 main()
 {   EvmConnection_t     conn;
     EvmStatus_t         status;
     fd_set              read_fds;
     int                 conn_fd;
     EvmBoolean_t        io_waiting;

     status = EvmConnCreate(EvmCONNECTION_LISTEN, EvmRESPONSE_CALLBACK,
                         NULL,EventCB,NULL,&conn);
     if (status != EvmERROR_NONE)
     {   fprintf(stderr,"Failed to create EVM listening connection\n");
         exit(1);
     }

     status = EvmConnSubscribe(conn,NULL,"[name sys.unix.evm.msg.user]");
     if (status != EvmERROR_NONE)
     {   fprintf(stderr,"Failed to subscribe for event notification\n");
         exit(1);
     }

     EvmConnFdGet(conn,&conn_fd); 1

     for (;;) 2
     {   FD_ZERO(&read_fds);
         FD_SET(fileno(stdin),&read_fds);
         FD_SET(conn_fd,&read_fds);

         select(FD_SETSIZE,&read_fds,NULL,NULL,NULL);

         if (FD_ISSET(fileno(stdin),&read_fds))
             HandleInput();

         status = EvmConnCheck(conn,&io_waiting); 3
         if (status != EvmERROR_NONE)
         {   fprintf(stderr,"Connection error\n");
             exit(1);
         }
         if (io_waiting)
         {   status = EvmConnDispatch(conn);
             if (status != EvmERROR_NONE)
             {   fprintf(stderr,"Connection dispatch error\n");
                 exit(1);
             }
         }
     }
}
/*================================================
 * Function: HandleInput()
 *================================================*/

void HandleInput() 4
{   char buff[256];
    if (feof(stdin))
        exit(0);

    if (fgets(buff,sizeof(buff),stdin) == NULL)
        exit(0);
```

Example 14–8: Handling Multiple I/O Sources (cont.)

```
    if (buff[0] == '\n')
        exit(0);

    fprintf(stdout,buff);
}
/*==============================================
 * Function: EventCB()
 *==============================================*/

void EventCB(EvmConnection_t conn, EvmCallbackArg_t cbarg,  5
                 EvmCallbackData_t *cbdata)
 {   char buff[256];

     switch (cbdata->reason) {
     case    EvmREASON_EVENT_DELIVERED:

             EvmEventFormat(buff, sizeof(buff), cbdata->event);
             fprintf(stdout,"Event: %s\n",buff);

             EvmEventDestroy(cbdata->event);
             break;
     default:
             break;
     }
 }
```

1 Use EvmConnFdGet( ) to find the file descriptor assigned to the connection.

2 In this example, select is used to wait for I/O activity because this allows waiting on multiple file descriptors. When the select call returns, check to see which of the file descriptors has activity, and deal with it appropriately.

3 EvmConnCheck( ) is used here to determine whether any activity is outstanding on the EVM connection. Because the connection's file descriptor is known, FD_ISSET( ) could have been used for the same purpose.

4 The HandleInput( ) function reads lines of input from stdin and echoes them to stdout. It terminates the program on error or if an empty line is read.

5 The event callback function is identical to the one used in the previous example.

14.7.12.8 Using Filter Evaluators

Some event subscribers need to monitor events using a variety of criteria, and then react in different ways according to the attributes of the incoming events. For example, the EVM logger reads a set of filter strings from its configuration file and subscribes for all of the events described in the strings. Then, as each event arrives, it has to determine which of the filter strings the event matches in order to write it to the correct set of logs.

One approach is to create multiple connections to the EVM daemon and subscribe on each connection with a separate filter string. However, this is costly in connection overhead and frequently results in the same event being sent across two or more of the connections.

A much better approach is to take all of the filter strings and combine them into a single logical string using the OR logical operator. This combined string is then used to subscribe for all matching events on a single connection. The combined string can then be discarded, but the original set of strings must be retained. It can be used to resubscribe later if the connection has to be re-established or if the filter has to change.

However, when an event arrives, you need to know which of the original filter strings it matches, so that you can decide what to do with it. You can use an EVM filter evaluator to do this. A filter evaluator is an object that can be created, loaded with a filter string, and then passed a series of events to determine which (if any) of the events match the filter. If you maintain a separate evaluator for each of the original filter strings, you can apply each incoming event to each of the evaluators to decide which evaluators match the event.

Example 14–9 demonstrates this technique by using three simple filter strings and by printing a different message according to which of the filters, if any, each incoming event matches.

The example introduces the following functions:

- `EvmFilterCreate` — Establishes an instance of a filter evaluator and returns a handle. (See `EvmFilterCreate`(3) and `EvmFilter`(5) for details.)
- `EvmFilterSet` — Passes a filter string to the filter evaluator to be used in subsequent matches. (See `EvmFilterSet`(3) and `EvmFilter`(5) for details.)
- `EvmFilterTest` — Compares a specified event with the filter string currently associated with the filter evaluator. If the event matches the filter string, `EvmFilterTest` returns `EvmTRUE`; otherwise, it returns `EvmFALSE`. (See `EvmFilterTest`(3) and `EvmFilter`(5) for details.)
- `EvmFilterDestroy` — Destroys a filter evaluator, freeing up all associated resources. (See `EvmFilterDestroy`(3) and `EvmFilter`(5) for details.)

Example 14–9: Using Filter Evaluators

```
#include <stdio.h>
#include <sys/time.h>
#include <evm/evm.h>

void EventCB(EvmConnection_t conn, EvmCallbackArg_t cbarg,
           EvmCallbackData_t *cbdata);

#define FILTER_1 "[name *.class_1]" 1
#define FILTER_2 "[name *.class_2]"
#define FILTER_3 "([name *.class_2] | [name *.class_3]) & [priority >= 300]"

/*===============================================
 * Function: main()
 *===============================================*/
 main()
 {   EvmConnection_t     conn;
     Evmstatus_t         status;
     int                 conn_fd;
     char                *filter_string;

     status = EvmConnCreate(EvmCONNECTION_LISTEN, EvmRESPONSE_CALLBACK,
                         NULL,EventCB,NULL,&conn);
     if (status != EvmERROR_NONE)
     {   fprintf(stderr,"Failed to create EVM listening connection\n");
         exit(1);
     }

     filter_string = (char *)malloc(strlen(FILTER_1) + strlen(FILTER_2) +
                      strlen(FILTER_3) + 30); 2
     sprintf(filter_string,"(%s) | (%s) | (%s)",FILTER_1,FILTER_2,FILTER_3);

     status = EvmConnSubscribe(conn,NULL,filter_string); 3
     if (status != EvmERROR_NONE)
     {   fprintf(stderr,"Failed to subscribe for event notification\n");
         exit(1);
     }
     free(filter_string);

     for (;;) 4
     {   status = EvmConnWait(conn,NULL);
         if (status != EvmERROR_NONE)
         {   fprintf(stderr,"Connection error\n");
             exit(1);
         }
         if (EvmConnDispatch(conn) != EvmERROR_NONE)
         {   fprintf(stderr,"Connection dispatch error\n");
             exit(1);
         }
      }
   }

/*===============================================
 * Function: EventCB()
 *===============================================*/
void EventCB(EvmConnection_t conn, EvmCallbackArg_t cbarg,
                EvmCallbackData_t *cbdata)
{   EvmBoolean_t        match;

    static EvmFilter_t  f1,f2,f3; 5
    static EvmBoolean_t filters_initialized = EvmFALSE;

    if (! filters_initialized)
```

Example 14–9: Using Filter Evaluators (cont.)

```
    {   if (EvmFilterCreate(&f1) != EvmERROR_NONE)
        {   fprintf(stderr,"Failed to create filter evaluator\n");
            exit(1);
        }
        if (EvmFilterSet(f1,FILTER_1) != EvmERROR_NONE)
        {   fprintf(stderr,"Failed to set filter evaluator\n");
            exit(1);
        }

        EvmFilterCreate(&f2);
        EvmFilterSet(f2,FILTER_2);
        EvmFilterCreate(&f3);
        EvmFilterSet(f3,FILTER_3);

        filters_initialized = EvmTRUE;
    }

    switch (cbdata->reason) { 6
    case    EvmREASON_EVENT_DELIVERED:

            EvmFilterTest(f1,cbdata->event,&match);
            if (match)
                fprintf(stdout,"Filter 1 event received\n");

            EvmFilterTest(f2,cbdata->event,&match);
            if (match)
                fprintf(stdout,"Filter 2 event received\n");

            EvmFilterTest(f3,cbdata->event,&match);
            if (match)
                fprintf(stdout,"Filter 3 event received\n");

            EvmEventDestroy(cbdata->event);
            break;

    default:
            break;
    }
}
```

1 This section of code defines three simple filter strings. The first two filter by name alone, but the third string selects events with either of two names, provided they have a priority of at least 300.

2 This section of code combines the three filter strings into a single logical expression, surrounding each substring with parentheses and separating them with the logical OR operator. This will cause the daemon to notify you of any event that matches at least one of the substrings.

3 Use the combined string to subscribe for the events, and then, because the string is no longer useful to you, free its space. If you have to resubscribe for any reason, you can always recombine the substrings.

4 After the evaluators have been established, keep them so that you can use them for all events that you receive (that is, make them static entities). Alternatively, you could create a new set of evaluators each time you receive an event, and destroy them as you exit the function. However, retaining them avoids the repeated setup and teardown overhead.

5 This section of code creates three filter evaluators using `EvmFilterCreate( )` and loads them with the three filter strings. For clarity, only the return status for the first filter is checked in this example, but in production code it is important to check it in each case. Note that this code is executed only the first time we receive an event.

6 This section of code tests each incoming event against each of the three filter evaluators established in the previous section, and prints a different message for each evaluator that is matched by the event. It is quite possible that some events will match more than one evaluator; in which case, more than one message is printed.

14.7.12.9 Matching Event Names

The EVM naming policy allows an event name to be extended with any number of trailing components, yet still match its base name. This means that you cannot depend on an event having exactly the name you expect it to have because it may have extra trailing components.

Therefore, when you need to compare an event's name against a known name, do not use the usual string comparison functions because they will incorrectly fail to match the name if components have been added. Instead, you should use EVM's name-matching functions. These functions match an event name against the pattern you supply, ignoring any trailing components in the candidate name. They also allow you to include wildcard characters in your name pattern.

Example 14–10 introduces the following function:

- `EvmEventNameMatch` — Takes the following input arguments: an event name string (which may contain wildcard characters) and an event. Returns an indication of whether the event matches the name string.

The following function is related to `EvmEventNameMatch`:

- `EvmEventNameMatchStr` — Takes an event name in a character string, rather than extracting it from an event.

Example 14–10: Matching Event Names

```
#include <stdio.h>
#include <evm/evm.h>

/*===============================================
 * Function: main()
 *===============================================*/
main()
{   EvmStatus_t     status;
    EvmEvent_t      event;
    EvmBoolean_t    match;
    char            buff[80];

    while (EvmERROR_NONE == EvmEventRead(fileno(stdin),&event)) 1
    {   EvmEventNameMatch("*.msg",event,&match);
        if (match)
        {   EvmEventFormat(buff,sizeof(buff),event);
            fprintf(stdout,"%s\n",buff);
        }
    }

}
```

1 This section of code reads events from `stdin` and displays only those events that match the wildcard string `*.msg`. This match will work even though events of this type usually have the name `sys.unix.evm.msg.user` or `sys.unix.evm.msg.admin`.

14.7.12.10 Dealing with Missed Events

If you are developing an application that subscribes for a set of events that could be posted in high volume over a short period, your program may occasionally be notified by the EVM daemon that it has missed one or more events. This is most likely to occur if the program has a significant amount of processing to do and might not be able to react immediately when a new event arrives, causing unhandled events to fill the connection buffers. See Section 14.7.10 for more information about missed events.

Example 14–11 subscribes for all events and displays each event on `stdout` as it arrives. It watches for notification of missed events by including the `EvmREASON_EVENTS_MISSED` reason in its callback function and displays a message that includes the number of events missed.

Example 14–11: Dealing with Missed Events

```
#include <stdio.h>
#include <evm/evm.h>

void EventCB(EvmConnection_t conn, EvmCallbackArg_t cbarg,
                      EvmCallbackData_t *cbdata);

/*===============================================
 * Function: main()
```

Example 14–11: Dealing with Missed Events (cont.)

```
 *===============================================*/
main()
{   EvmConnection_t     conn;
    EvmStatus_t         status;
    int                 conn_fd;

    status = EvmConnCreate(EvmCONNECTION_LISTEN, EvmRESPONSE_CALLBACK,
                        NULL,EventCB,NULL,&conn);
    if (status != EvmERROR_NONE)
    {   fprintf(stderr,"Failed to create EVM listening connection\n");
        exit(1);
    }

    status = EvmConnSubscribe(conn,NULL,"[name *]");
    if (status != EvmERROR_NONE)
    {   fprintf(stderr,"Failed to subscribe for event notification\n"); 1
        exit(1);
    }

    for (;;)
    {   status = EvmConnWait(conn,NULL);
        if (status != EvmERROR_NONE)
        {   fprintf(stderr,"Connection error\n");
            exit(1);
        }
        if (EvmConnDispatch(conn) != EvmERROR_NONE)
        {   fprintf(stderr,"Connection dispatch error\n");
            exit(1);
        }
    }
}

/*===============================================
 * Function: EventCB()
 *===============================================*/
 void
 EventCB(EvmConnection_t conn, EvmCallbackArg_t cbarg,
                 EvmCallbackData_t *cbdata)
 {   char                buff[256];

     switch (cbdata->reason) {
     case    EvmREASON_EVENT_DELIVERED:
             EvmEventFormat(buff, sizeof(buff), cbdata->event);
             fprintf(stdout,"Event: %s\n",buff);

             EvmEventDestroy(cbdata->event);

             sleep(1); 2
             break;

     case    EvmREASON_EVENTS_MISSED: 3
             fprintf(stdout,"*** Missed %d incoming events\n",
                 cbdata->extension.eventMissedData.missedCount);
             break;
     default:
```

Example 14–11: Dealing with Missed Events (cont.)

```
             break;
     }
}
```

1 This example subscribes for notification of all events.

2 To demonstrate missing events, sleep for one second after each event is received. This simulates a heavy processing load and ensures that the input connection buffer will be filled if the event load is heavy.

3 The callback function will be invoked with reason code `EvmREASON_EVENTS_MISSED` if the daemon has been unable to send one or more missed events. The callback data member `extension.eventMissedData.missedCount` contains a count of the number of missed events.

14.8 Adding an Event Channel to EVM

The term **event channel** describes any facility used to publish or retrieve event information, and might refer to any of the following:

- Simple log file
- Event management system
- Program that can be run to obtain a snapshot of status information

An event channel may be an *active* channel, in which case it posts its own event information to EVM as soon as the event occurs, or it may be a *passive* channel, meaning that the information may accumulate within the channel but EVM has to take the action to look for it.

An event channel does not have to be a formal event notification mechanism. It can be anything that is capable of storing information or showing a status or a change of state. For example:

- If an application or system components writes state information to its own log file, each entry in the log file can be thought of as an event. The log file can be checked periodically for new entries, and an event can be posted for each new entry that is found. The log file itself also serves as a place from which to retrieve historical events.
- Some applications and system components provide a means of querying status. In these cases, the status can be monitored periodically, and an event can be posted whenever a significant change is detected. Because this type of event channel typically does not include a means of storing event information, EVM's own logging mechanism can be used to log the events.

- If a central part of an application handles all event information for the application, it may be feasible to modify the handling code so that it also forwards event information to EVM. This is an example of an *active* event channel.

The process of taking an existing event channel and making it accessible through EVM is termed **encapsulation**.

EVM event channels are configured through the channel configuration file. This file is read by the EVM channel manager when EVM is started, and is also used by command-line utilities when channel information is needed. When you change this file, you must make the channel manager aware of the change by entering the following command:

```
evmreload -c
```

See `evmchannel.conf`(4) for the syntax of the channel configuration file. You can encapsulate an event channel by providing a set of executable programs to handle the various channel functions. The functions are:

- The get function — Retrieves historical events from the channel when `evmget` is run.
- The details function — Provides a detailed display of an event stream when `evmshow` is run with the `-d` option.
- The explain function — Provides explanatory text for an event stream when `evmshow` is run with the `-x` option. This function is also typically invoked by the channel's details function.
- The monitor function — Executed periodically by the EVM channel manager to check the channel status and post events if necessary.
- The cleanup function — Executed daily by the EVM channel manager to perform any cleanup actions necessary for the channel.

Each of these functions is optional, and is defined for the channel by including the appropriate entry in a channel definition in the channel configuration file. Channel functions can be any kind of executable file and must operate as described in the sections that follow this section.

Channel functions that use temporary files should be sure to clean up before exiting, and they must also be able to clean up if they are interrupted.

You must also:

- Add event templates for any events which may be posted or retrieved through a new event channel. See Section 14.6.3 for information about adding event templates.
- Make sure that the events are given the correct posting and access privileges, by modifying the EVM authorization file as necessary. See

the *System Administration* manual for details of how to control access to events.

14.8.1 The Get Function

The channel get function is executed by the EVM `get_server`, which is executed by the EVM daemon to handle event retrieval requests made through `evmget`. This function is always executed as root and must take appropriate security precautions.

The function must support the following invocation syntax:

`function-name` [**`-f`** *`filter-string`*]

If desired, other arguments may be passed to the function by including them in the function's line in the channel configuration file.

When executed, the get function should retrieve events from the channel's log files, convert them to EVM event format, and write the EVM events to its `stdout` stream. If a *`filter-string`* is supplied, only events that match the filter can be written to `stdout`. Error messages can be written to `stderr` and will be passed back to `evmget` for output to its `stderr` stream, so be sure that they are clearly identified as originating in this function. Do not write anything other than EVM events to `stdout`.

The form of a get script depends very much on the form in which the native events are stored. In general, the steps will be:

1. Use standard UNIX tools such as `grep`, `awk`, and `sed`, or a programming language such as perl, to select the event lines, removing blank lines and comments, and reformat them as necessary for the next step. This should be a reasonably simple matter if the events are single lines of text, with a constant format in each line, and include items such as a timestamp, host name, and message in the same position in every line.
2. Convert the lines into EVM events. You may be able to do this by using UNIX tools to format the lines into a form suitable for input to `evmpost`, using the `-r` option to produce EVM events on `stdout` instead of posting them. Alternatively, for a faster conversion, you can use the EVM channel utility `/usr/share/evm/channels/bin/text2evm` to do the conversion. This tool currently requires input of the form:

   `evm-event-name date time host user message`

   where:

   - *`evm-event-name`* is the `NAME` of the EVM event that will be produced by the tool. The first few components of the names of all events passed through a particular channel should be the same, so that the events can be associated with the channel.

- *`date`* and *`time`* make up the `TIMESTAMP` item for the event. The components of the date must be in the basic format *`year/month/day`*, where the slash (`/`) characters can be replaced by any of the following characters:

  hyphen (`-`), colon (`:`), period (`.`)

  There is also some flexibility in the format of the values to allow for the variations in logfile formats and to minimize the amount of conversion work that may need to be done in a shell script:

  - The year can be two or four digits, or can be replaced with the question mark (`?`) character. If four digits are supplied, the year is taken unchanged. If two digits are supplied, and the value is 90 or greater, 1900 is added to the value to give the year. If the value is less than 90, 2000 is added to it. If a `?` character is supplied instead of a year, the tool calculates the implied year by comparing the current date with the month and day fields in the supplied timestamp. If the timestamp month and day represent a date that is later than the current date, the year value defaults to the previous year; otherwise, it is assumed to be the current year.
  - The month may be either a two-digit month in the range 1-12 or an alphabetic month name in either abbreviated (for example, Feb) or unabbreviated form (February). The month name may be supplied either in English or in the language of the system's default locale.

  The time must be in the format *`hours`*`:`*`minutes`*`:`*`seconds`*.
- *`host`* is the `HOST_NAME` item, and can be determined with the `hostname` command.
- *`user`* is the `USER_NAME` item. If all entries in the log are written by a single application program or subsystem, it is probably appropriate to set the user field to the owner of the logfile.
- *`message`* is the message text for the event, and is inserted into the event as the `FORMAT` data item.

3. If a *filter-string* was supplied, pass the output through `evmshow`, using the `-f` and `-r` options, to restrict the output to that requested in the filter.
4. Finally, if you want the retrieved events to include data items contained in the events' templates, you can pipe the output through the EVM channel utility, `/usr/share/evm/channels/bin/merge_template`. This program reads EVM events on `stdin` and obtains the corresponding template for each from the EVM daemon. It then merges

the template information into the event and writes the resulting expanded event to stdout.

If your channel's log files are difficult to convert to EVM format — for example, because each entry is made up of multiple unstructured lines of text, which cannot be parsed easily — it may be better not to supply a get function, but instead to allow the events to be logged by the EVM logger as they are posted. This consumes more storage space because the events would be stored in two places, but it may significantly improve retrieval time and programming effort.

If you supply your own get function for the channel, be sure to change the filter strings in the EVM logger's configuration file so that your events are not duplicated in the EVM log. See the *System Administration* manual for information on how to change the logger's configuration file.

14.8.2 The Details Function

The details function is executed by `evmshow` when it is invoked with the `-d` option. Although it is currently executed with the privileges of the user executing `evmshow`, this is likely to change in a future release, so it is important that it take appropriate security precautions.

The function must support the following invocation syntax:

function-name [*arguments*]

If arguments are specified in the details function line in the channel configuration file, they are passed directly to the function when it is executed. The details function must accept a stream of EVM events through `stdin`, and display on `stdout` a stream of text that describes the contents of each event. Various forms of `evmshow` (notably `-x` and `-D`) are likely to be useful in producing the output, but be careful not to use the `-d` option because that would result in a recursive loop.

Unless redirected, messages written to `stderr` will appear on `evmshow`'s `stderr` stream, so if needed they should be clearly identified as being written by this function.

You can use the `evmlog` channel function, `/usr/share/evm/channels/evmlog/evmlog_details` as a model for developing a details script for a new channel. If you do not supply a details function, `evmshow` and the event viewer display a formatted dump of the contents of any event belonging to the channel whenever a detailed view is requested.

14.8.3 The Explain Function

The explain function is executed by `evmshow` when it is invoked with the `-x` option. Although it is currently executed with the privileges of the user executing `evmshow`, this is likely to change in a future release, so it is important that it take appropriate security precautions. The function must support the following invocation syntax:

`function-name event-name` [*`reference`*]

The explain function is invoked with the name of the event requiring explanation and an optional reference value. If supplied, the reference is the contents of the event's reference data item. If no reference is available, `evmshow` passes a hyphen (-) for this argument, but the function should also allow the argument to be omitted.

The explain function should use its arguments to produce a formatted explanation for the supplied event name, and write it as lines of text to `stdout`. If no explanation can be found, an appropriate message should be written to `stdout` in place of the explanation. Unless redirected, messages written to `stderr` will appear on `evmshow`'s `stderr` stream, so, if they are needed, they should be clearly identified as being written by this function

Your explain function can invoke the `evmlog` explain function, `/usr/share/evm/channels/evmlog/evmlog_explain`, provided that:

- In addition to the *`event-name`* argument, you pass a reference argument of the form `cat:`*`catalog-name`*[`:`*`set_number`*], where *`catalog-name`* is the name of an I18N catalog file containing the explanations for your channel's events and the optional *`set_number`* is the number of the message set containing the explanation, for example, `cat:myprod.cat:3`.
- Each explanation in the catalog begins with the name of an event enclosed in braces, for example, `{myco.myprod.myapp.startup}`.

The message catalog must be located according to normal I18N rules. To minimize search time, group the explanations into sets and provide the set numbers in the reference data items of the events. See `mkcatdefs`(1) and `gencat`(1) for the procedures to generate a catalog file.

14.8.4 The Monitor Function

The monitor function is executed by the EVM channel manager. It is executed with an `-init` argument when the channel manager is started and each time the channel manager is reconfigured with `evmreload`, and without the `-init` argument periodically thereafter. The execution period is controlled with the *`mon_period`* channel value. This function is always executed as root and must take appropriate security precautions.

The function must support the following invocation syntax:

function-name [**init**]

You can use the presence or absence of the init argument to decide whether the function needs to initialize any work files that it has to maintain. If desired, you can pass additional arguments to the function by including them on the command line in the channel configuration file, but note that the init argument is always passed as the *last* argument.

There are no restrictions on the actions that can be taken by the monitor function, although its job should generally be to check status and post events if it detects a change of state.

The function is invoked with no stdout and with stderr assigned to the channel manager's log file. If the function does not reassign stderr, any messages written to it should have the same format as the channel manager's log entries and should be clearly identifiable as coming from your channel. Alternatively, you can post error messages as EVM events, taking care not to report the same condition unnecessarily each time the function in invoked.

The following example monitor script initializes itself by counting the number of lines in a log file and saving the count in a state file. On subsequent invocations, it compares the number of lines in the file with the previous number, extracts each new line with the UNIX tail command, and posts it as an EVM event with evmpost.

```
#! /bin/sh
INIT=$1
STATE=/tmp/mylog.state
LOG=/tmp/mylog

EVENT_NAME=myco.admin.mylog.line_added

# No log?  Create one!
if [ ! -f $LOG ]
then
    touch $LOG
fi

# If we're initializing then save the current logfile
# state and exit:
if [ "$INIT" != "" ]
then
    # Count the lines in the demolog, and save the count
    # in the state file:
    wc -l $LOG | awk '{print $1}' > $STATE
    exit
fi

# Find out how many lines there were in the file last time
# we checked:
OLDCOUNT=`cat $STATE`

# How many now?
```

```
NEWCOUNT=‘wc -l $LOG | awk ’{print $1}’‘
if [ $NEWCOUNT > $OLDCOUNT ]
then
    # Save the new line count to the state file:
    echo $NEWCOUNT > $STATE

    # What’s the difference between the old and new counts?
    diff=‘expr $NEWCOUNT - $OLDCOUNT | awk ’{print $1}’‘

    # Post an event for each new line:
    tail -$diff $LOG | while read LINE
    do
        echo ’event { name ’${EVENT_NAME} \
            ’ var {name msg type STRING value "’$LINE’"} }’ | evmpost
    done
fi
```

14.8.5 The Cleanup Function

The cleanup function is executed daily by the EVM channel manager, at the time specified in the channel configuration file, to perform housekeeping actions such as archiving and deleting the channel’s log files. This function is always executed as root and must take appropriate security precautions.

The cleanup function is specified as a command line that is executed exactly as specified, so arguments may be passed to the function through the command line if desired. The function is free to take whatever action is appropriate. It is executed with no `stdout` and with `stderr` assigned to the channel manager’s log file, so any desired status messages should generally be issued in the form of EVM events by using `evmpost`, instead of being written to `stderr`. If the function does not reassign `stderr`, any messages written to it must have the same format as those written by the channel manager and must be clearly identifiable as coming from your channel. Nothing should be written to `stdout`.

Write the function so that it has the same effect regardless of the time of day at which it is run; for example, it might use the `-mtime` option of the `find` command to identify log files to be archived.

14.8.6 Channel Security

In most cases, channel functions are executed by processes that are children of the EVM daemon, and, as a result, they will be run with full root privileges. Because of this, you must protect your system’s integrity by ensuring that:

- Functions are placed in a directory that has restricted write privileges.
- Functions themselves have restricted write and execute privileges.
- Functions do not call other programs that have inappropriate privileges.

A

Using 32-Bit Pointers on Tru64 UNIX Systems

The size of the default pointer type on Tru64 UNIX systems is 64 bits, and all system interfaces use 64-bit pointers. The Compaq C compiler, in addition to supporting 64-bit pointers, also supports the use of 32-bit pointers.

In most cases, 64-bit pointers do not cause any problems in a program and the issue of 32-bit pointers can be ignored altogether. However, in the following cases, the issue of 32-bit pointers does become a concern:

- If you are porting a program with pointer-to-`int` assignments
- If the 64-bit pointer size is unnecessary for your program and it causes your program to use too much memory — for example, your program uses very large structures composed of pointer fields and has no need to exceed the 32-bit address range in those fields
- If you are developing a program for a mixed system environment (32- and 64-bit systems) and the program's in-memory structures are accessed from both types of systems

The use of 32-bit pointers in applications requires that the applications be compiled and linked with special options and, depending on the specific nature of the code, may also require source-code changes.

The following types of pointers are referred to in this appendix:

- Short pointer: A 32-bit pointer. When a short pointer is declared, 32 bits are allocated.
- Long pointer: A 64-bit pointer. When a long pointer is declared, 64 bits are allocated. This is the default pointer type on Tru64 UNIX systems.
- Simple pointer: A pointer to a nonpointer data type, for example, `int *num_val`. More specifically, the pointed-to type contains no pointers, so the size of the pointed-to type does not depend on the size of a pointer.
- Compound pointer: A pointer to a data type whose size depends upon the size of a pointer, for example, `char **FontList`.

A.1 Compiler-System and Language Support for 32-Bit Pointers

The following mechanisms control the use of 32-bit pointers:

- The cc options -xtaso and -xtaso_short are needed for compiling programs that use pointers with a 32-bit data type:
  - –xtaso

    Enables recognition of the #pragma pointer_size directive and causes -taso (truncated address support option) to be passed to the linker (if linking).
  - –xtaso_short

    Same as -xtaso, except -xtaso_short also sets the initial compiler default state to use short pointers. Because all system routines continue to use 64-bit pointers, most applications require source changes when compiled in this way. However, the use of protect_headers_setup (see Section A.3.3) can greatly reduce or eliminate the need to change the source code.

    Because the use of short pointers, in general, requires a thorough knowledge of the application they are applied to, -xtaso_short is not recommended for use as a porting aid. In particular, do not attempt to use -xtaso_short to port a poorly written program (that is, a program that heavily mixes pointer and int values).
- The ld option -taso ensures that executable files and associated shared libraries are located in the lower 31-bit addressable virtual address space. The -taso option can be helpful when porting programs that assume address values can be stored in 32-bit variables (that is, programs that assume that pointers are the same length as int variables). The -taso option does not affect the size of the pointer data type; it just ensures that the value of any pointer in a program would be the same in either a 32-bit or a 64-bit representation.

  The –taso linker option does impose restrictions on the run-time environment and how libraries can be used. See Section A.2 for details on the -taso option.
- The #pragma pointer_size directive controls the size of pointer types in a C program. These pragmas are recognized by the compiler only if the –xtaso or –xtaso_short options are specified on the cc command line; they are ignored if neither of the options is specified.

  See Section 3.12 for details on the #pragma pointer_size directive.

The following example demonstrates the use of both short and long pointers:

```
main ()
```

```
  {
     int *a_ptr;

     printf ("A pointer is %ld bytes\n", sizeof (a_ptr));
  }
```

When compiled with either the default settings or the –`xtaso` option, the sample program prints the following message:

```
A pointer is 8 bytes
```

When compiled with the –`xtaso_short` option, this sample program prints the following message:

```
A pointer is 4 bytes
```

A.2 Using the -taso Option

The `-taso` option establishes 32-bit addressing within all 64-bit pointers within a program. It thereby solves almost all 32-bit addressing problems, except those that require constraining the physical size of some pointers to 32-bits (which is handled by the `-xtaso` or `-xtaso_short` option and `pointer_size` pragmas).

The `-taso` option is most frequently used to handle addressing problems introduced by pointer-to-`int` assignments in a program. Many C programs, especially older C programs that do not conform to currently accepted programming practices, assign pointers to `int` variables. Such assignments are not recommended, but they do produce correct results on systems in which pointers and `int` variables are the same size. However, on a Tru64 UNIX system, this practice can produce incorrect results because the high-order 32 bits of an address are lost when a 64-bit pointer is assigned to a 32-bit `int` variable. The following code fragment illustrates this problem:

```
{
char *x;   /* 64-bit pointer */
int z;     /* 32-bit int variable */
.
.
.
    x = malloc(1024); /* get memory and store address in 64 bits */
    z = x;    /* assign low-order 32 bits of 64-bit pointer to
                 32-bit int variable */
}
```

The most portable way to fix the problem presented by pointer-to-`int` assignments in existing source code is to modify the code to eliminate this type of assignment. However, in the case of large applications, this can be time-consuming. (To find pointer-to-`int` assignments in existing source code, use the `lint -Q` command.)

Another way to overcome this problem is to use the -taso option. The -taso option makes it unnecessary for the pointer-to-int assignments to be modified. It does this by causing a program's address space to be arranged so that all locations within the program — when it starts execution — can be expressed within the 31 low-order bits of a 64-bit address, including the addresses of routines and data coming from shared libraries.

The -taso option does not in any way affect the sizes used for any of the data types supported by the system. Its only effect on any of the data types is to limit addresses in pointers to 31 bits (that is, the size of pointers remains at 64 bits, but addresses use only the low-order 31 bits).

A.2.1 Use and Effects of the -taso Option

The -taso option can be specified on the cc or ld command lines used to create object modules. (If specified on the cc command line, the option is passed to the ld linker.) The -taso option directs the linker to set a flag in object modules, and this flag directs the loader to load the modules into 31-bit address space.

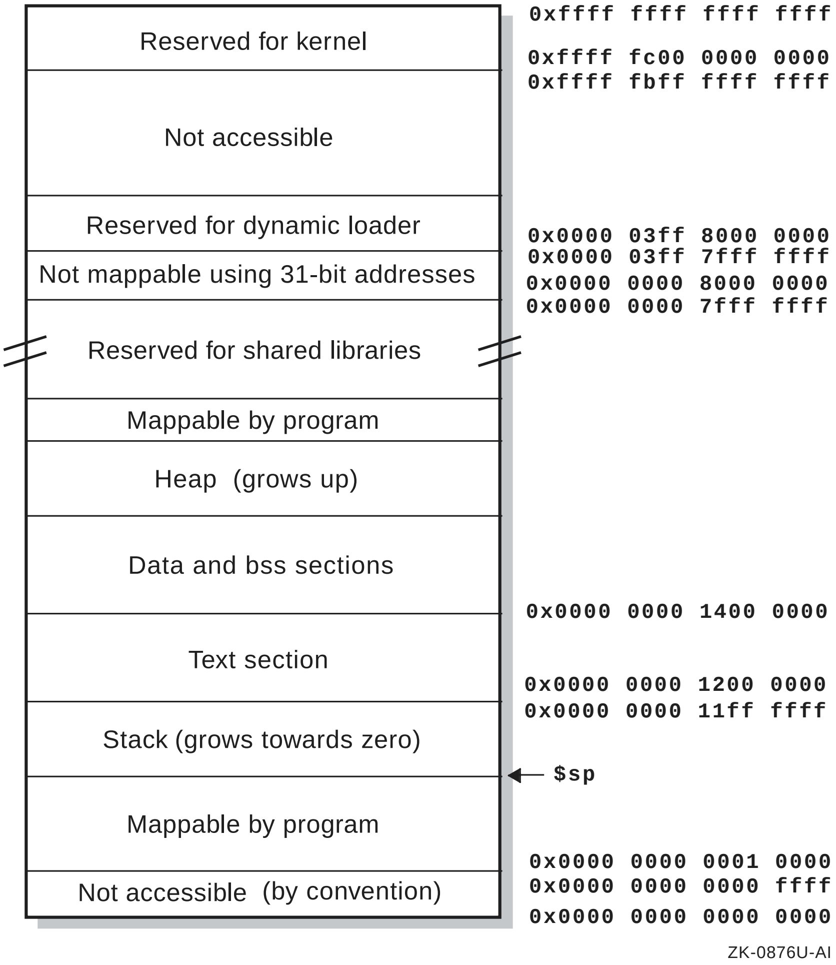
The 31-bit address limit is used to avoid the possibility of setting the sign bit (bit 31) in 32-bit int variables when pointer-to-int assignments are made. Allowing the sign bit to be set in an int variable by pointer-to-int assignment would create a potential problem with sign extension. For example:

```
{
char *x;   /* 64-bit pointer */
int z;     /* 32-bit int variable */
   .
   .
   .
           /* address of named_obj = 0x0000 0000 8000 0000 */
      x = &named_obj;  /* 0x0000 0000 8000 0000 = original pointer
                          value */
      z = x;  /* 0x8000 0000 = value created by pointer-to-int
                 assignment */
      x = z;  /* 0xffff ffff 8000 0000 = value created by pointer-
                 to-int-to-pointer or pointer-to-int-to-long
                 assignment (32 high-order bits set to ones by
                 sign extension) */
}
```

The -taso option ensures that the text and data segments of an application are loaded into memory that can be reached by a 31-bit address. Therefore, whenever a pointer is assigned to an int variable, the values of the 64-bit pointer and the 32-bit variable will always be identical (except in the special situations described in Section A.2.2).

Figure A–1 is an example of a memory diagram of programs that use the -taso and -call_shared options. (If you invoke the linker through the cc command, the default is -call_shared. If you invoke ld directly, the default is -non_shared.)

Figure A–1: Layout of Memory Under -taso Option



Note that stack and heap addresses will also fit into 31 bits. The stack grows downward from the bottom of the text segment, and the heap grows upward from the top of the data segment.

The -T and -D options (linker options that are used to set text and data segment addresses, respectively) can also be used to ensure that the text and data segments of an application are loaded into low memory. The -taso option, however, in addition to setting default addresses for text and data segments, also causes shared libraries linked outside the 31-bit address space to be appropriately relocated by the loader.

The default addresses used for the text and data segments are determined by the options that you specify on the cc command line:

- Specifying the -non_shared or -call_shared option with the -taso option results in the following defaults:

  0x0000 0000 1200 0000 (text segment's starting address)
  0x0000 0000 1400 0000 (data segment's starting address)

- Specifying the -shared option with the -taso option results in the following defaults:

  0x0000 0000 7000 0000 (text segment's starting address)
  0x0000 0000 8000 0000 (data segment's ending address)

Using these default values produces sufficient amounts of space for text and data segments for most applications (see the *Object File and Symbol Table Format Specification* for details on the contents of text and data segments). The default values also allow an application to allocate a large amount of mmap space.

If you specify the -taso option and also specify text and data segment address values with -T and -D , the values specified override the -taso default addresses.

You can use the odump utility to check whether a program was built successfully within 31-bit address space. To display the start addresses of the text, data, and bss segments, enter the following command:

```
% odump -ov obj_file_x.o
```

None of the addresses should have any bits set in bits 31 to 63; only bits 0 to 30 should ever be set.

Shared objects built with -taso cannot be linked with shared objects that were not built with -taso. If you attempt to link taso shared objects with nontaso shared objects, the following error message is displayed:

```
Cannot mix 32 and 64 bit shared objects without -taso option
```

A.2.2 Limits on the Effects of the -taso Option

The -taso option does not prevent a program from mapping addresses outside the 31-bit limit, and it does not issue warning messages if this is done. Such addresses could be established using any one of the following mechanisms:

- **-T and -D options**

  As noted previously, if the -T and -D options are used with the -taso option, the values that you specify for them will override the -taso option's default values. Therefore, to avoid defeating the purpose of the

-taso option, you must select addresses for the -T and -D options that are within the 31-bit address range.

- **malloc() function**

  To avoid problems with addressing when you use malloc in a taso application, provide a declaration in your application that preempts libc's __taso_mode variable, a global read-only "unsigned long" variable, with one that is initialized to a static value of 1. This value tells the malloc routine that it is working with a taso application, and malloc will then check to ensure that the address of any allocated piece of memory is within the 31–bit range.

- **mmap system call**

  Taso applications that use the mmap system call must use a jacket routine to mmap to ensure that mapping addresses do not exceed a 31-bit range. This entails taking the following steps:

  1. To prevent mmap from allocating space outside the 31-bit address space, specify the following compilation option on the cc command line for all modules (or at least all modules that refer to mmap):

     ```
     -Dmmap=_mmap_32_
     ```

     This option equates the name of the mmap function with the name of a jacket routine (_mmap_32_). As a result, the jacket routine is invoked whenever references are made to the mmap function in the source program.

  2. If the mmap function is invoked in only one of the source modules, either include the jacket routine in that module or create an mmap_32.o object module and specify it on the cc command line. (The file specification for the jacket routine is /usr/opt/alt/usr/lib/support/mmap_32.c, and the file resides in the optional software subset CMPDEVENH*nnn*.)

     If the mmap function is invoked from more than one source file, you must use the method of creating an mmap_32.o object module and specifying it on a cc command line because including the jacket routine in more than one module would generate linker errors.

A.2.3 Behavior of malloc in the Taso Environment

Because of the way in which the loader sets up the memory space for a call-shared taso application (see Figure A–1), it is impossible to allocate more than 256M bytes of memory with the malloc( ) function unless special steps are taken.

To overcome this problem, you need to make more space available for allocations. This can be done in the following ways:

- Link -non_shared.

  The most effective solution for large allocations in a taso application is to link the application without shared libraries. If it is possible to link -non_shared, malloc( ) can allocate memory from the end of bss to 0x7fff ffff, assuming that the program does not use mmap to allocate memory somewhere in that range. (The problem with conflicting allocations associated with mmap is explained in the text for the next item in this list.)

- Use the malloc function's __sbrk_override tuning variable.

  The malloc routines (malloc, calloc, realloc, and valloc) normally allocate memory by means of the sbrk system call. However, an sbrk allocation will fail when sbrk encounters the text section of a shared library or a region allocated by means of mmap.

  Using the __sbrk_override tuning variable allows allocations to take place at any point in memory between 0x0001 0000 and 0x7fff ffff. If this variable is set, malloc will switch over to using mmap when it detects a failed call to sbrk, and mmap will satisfy the allocation request wherever it finds adequate space in the previously cited address range.

A.3 Using the –xtaso or –xtaso_short Option

The -xtaso and -xtaso_short options enable you to use both short (32-bit) and long (64-bit) pointer data types in a program. Note that the 64-bit data type is constrained to 31-bit addressing because -xtaso and -xtaso_short both engage the -taso option.

You should only use the -xtaso or -xtaso_short option when you have a need to use the short pointer data type in your program. If you want to use 32-bit addressing but do not need short pointers, use the -taso option.

Most programs that use short pointers will also need to use long pointers because Tru64 UNIX is a 64-bit system and all applications must use 64-bit pointers wherever pointer data is exchanged with the operating system or any system-supplied library. Because normal applications use the standard system data types, no conversion of pointers is needed. In an application that uses short pointers, it may be necessary to explicitly convert some short pointers to long pointers by using pointer_size pragmas (see Section 3.12).

Note

New applications for which the use of short pointers is being considered should be developed with long pointers first and then analyzed to determine whether short pointers would be beneficial.

A.3.1 Coding Considerations Associated with Changing Pointer Sizes

The following coding considerations may be pertinent when you are working with pointers in your source code:

- The size of pointers used in a `typedef` that includes pointers as part of its definition is determined when the `typedef` is declared, not when it is used. Therefore, if a short pointer is declared as part of a `typedef` definition, all variables that are declared using that `typedef` will use a short pointer, even if those variables are compiled in a context where long pointers are being declared.
- The size of pointers within a macro is governed by the context in which the macro is expanded. The only way to specify pointer size as part of a macro is by using a `typedef` declared with the desired pointer size.
- In general, conversions between short and long *simple* pointers are safe and are performed implicitly without the need for casts on assignments or function calls. On the other hand, *compound* pointers generally require source code changes to accommodate conversions between short and long pointer representations.

  For example, the argument vector, `argv`, is a long compound pointer and must be declared as such. Many X11 library functions return long compound pointers; the return values for these functions must be declared correctly or erroneous behavior will result. If a function was compiled to exclusively use short pointers and needed to access such a vector, it would be necessary to add code to copy the values from the long pointer vector into a short pointer vector before passing it to the function.
- Only the C and C++ compilers support the use of short pointers. Short pointers should not be passed from C and C++ routines to routines written in other languages.
- The `pointer_size` pragma and the `-xtaso_short` option have no effect on the size of the second argument to `main()`, traditionally called `argv`. This argument always has a size of 8 bytes even if the `pointer_size` pragma has been used to set other pointer sizes to 4 bytes.

A.3.2 Restrictions on the Use of 32-Bit Pointers

Most applications on Tru64 UNIX systems use addresses that are not representable in 32 bits. Therefore, no library that might be called by normal applications can contain short pointers. Vendors who create software libraries generally should not use short pointers.

A.3.3 Avoiding Problems with System Header Files

When the system libraries are built, the compiler assumes that pointers are 64 bits and that structure members are naturally aligned. These are the C and C++ compiler defaults. The interfaces to the system libraries (the header files in the `/usr/include` tree) do not explicitly encode these assumptions.

You can alter the compiler's assumptions about pointer size (with `-xtaso_short`) and structure member alignment (with `-Zp`*n* [where *n*!=8] or `-nomember_alignment`). If you use any of these options and your application includes a header file from the `/usr/include` tree and then calls a library function or uses types declared in that header file, problems may occur. In particular, the data layouts computed by the compiler when it processes the system header file declarations may differ from the layouts compiled into the system libraries. This situation can cause unpredictable results.

Run the script `protect_headers_setup.sh` immediately after the compiler is installed on your system to eliminate the possibility of problems with pointer size and data alignment under the conditions described in this section. See `protect_headers_setup`(8) for details on how and why this is done.

You can enable or disable the protection established by the `protect_headers_setup` script by using variations of the `-protect_headers` option on your compilation command line. See `cc`(1) for information about the `-protect_headers` option.

B

Differences in the System V Habitat

This appendix describes how to achieve source code compatibility for C language programs in the System V habitat. In addition, it provides a summary of system calls and library functions that differ from the default operating system.

B.1 Source Code Compatibility

To achieve source code compatibility for the C language programs, alter your shell's `PATH` environment variable and then compile and link your applications.

When you modify the `PATH` environment variable, access to the System V habitat works on two levels:

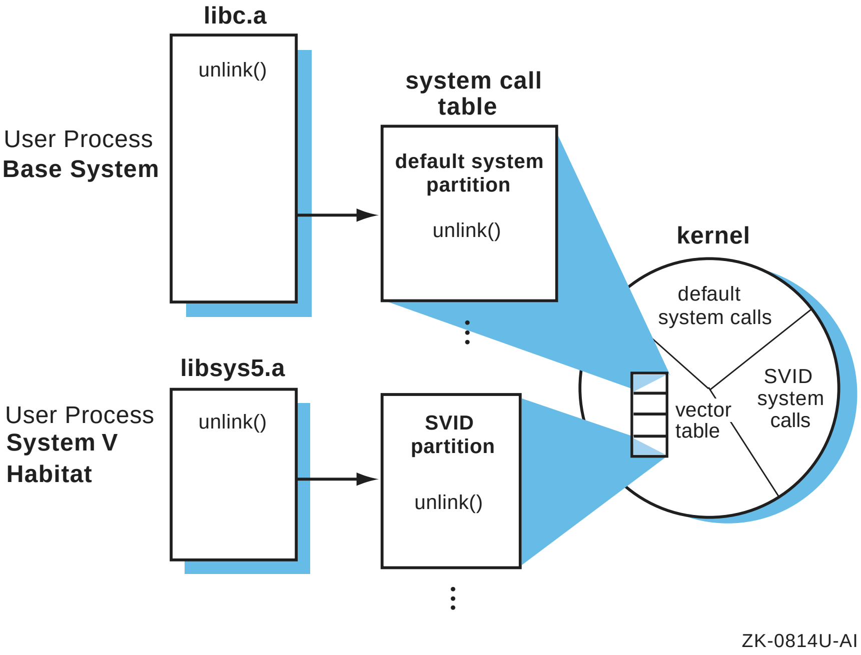
- The first level results from the modified `PATH` environment variable causing the System V versions of several user commands to execute instead of the default system versions.
- The second level results from executing the System V `cc` or `ld` commands.

Executing the System V versions of the `cc` and `ld` commands causes source code references to system calls and subroutines to be resolved against the libraries in the System V habitat. If a subroutine or system call is not found in the System V habitat, the reference is resolved against the standard default libraries and other libraries that you can specify with the commands. Also, the include file search path is altered so that the System V versions of the system header files (for example, `/usr/include` files) are used instead of the standard versions.

The library functions that invoke system calls use the system call table to locate the system primitives in the kernel. The base operating system contains several system call tables, including one for System V. The system calls that exhibit System V behavior have entries in the System V partition of the system call table.

When you link your program and your `PATH` is set for the System V habitat, `libsys5` is searched to resolve references to system calls. As shown in Figure B–1, the `unlink()` system call invoked by `libsys5` points to an entry in the System V partition of the system call table. This maps to a different area of the kernel than the mapping for the default system `unlink()` system call.

Figure B–1: System Call Resolution



The `cc` and `ld` commands that reside in the System V habitat are shell scripts that, when specified, add several options to the default system `cc` and `ld` commands before the commands are executed.

The `cc` command automatically inserts the `-I`*`path`* option on the command line to specify the use of the SVID versions of system header files. For example, the `/usr/include` file is used instead of the default version. System header files that do not have SVID differences are obtained from the default location.

The `cc` and `ld` commands automatically include the following options:

- The `-L`*`path`* option provides the path of the System V libraries.
- The `-lsys5` option indicates that the `libsys5.a` library should be searched before the standard C library to resolve system call and subroutine references.
- The `-D__SVID__` option selectively turns on SVID specific behavior from the default system.

By default, cc dynamically links programs using shared libraries when they exist. The System V habitat provides libsys5.so in addition to libsys5.a to support this feature.

The System V version of the cc and ld commands pass all user-specified command-line options to the default system versions of the cc and ld commands. This allows you to create library hierarchies. For example, if your PATH environment variable is set to the System V habitat and your program includes references to math library functions and libloc.a functions located in the /local/lib directory, you can compile the program as follows:

```
% cc –non_shared –L/local/lib src.c –lm –lloc
```

The System V cc command takes the preceding command line and adds the necessary options to search the System V habitat libraries, which are searched prior to the default libraries. It also includes any existing System V header files instead of the standard header files for /usr/include. Hence, if your environment is set to SVID 2, the preceding command line is processed as follows:

```
/bin/cc -D__SVID__ -I$SVID2PATH/usr/include -L$SVID2PATH/usr/lib \
–non_shared –L/local/lib src.c –lm –lloc –lsys5
```

Using this command line, libraries are searched in the following order:

1. /usr/lib/libm.a
2. /local/lib/libloc.a
3. SVID2PATH/usr/lib/libsys5.a
4. /usr/lib/libc.a

The libraries that are searched and the order that they are searched in depends on the function you are performing. For more information, see cc(1) and ld(1).

B.2 Summary of System Calls and Library Routines

Table B–1 describes the behavior of the system calls in the System V habitat. For a complete explanation of these system calls, refer to the reference pages for each system call. Table B–2 describes the behavior of the library functions in the System V habitat.

See the reference pages for complete descriptions of the system calls and library routines.

Table B–1: System Call Summary

System Call	System V Behavior
longjmp(2) and setjmp(2)	Saves and restores the stack only.
mknod(2)	Provides the ability to create a directory, regular file, or special file.
mount(2sv) and umount(2sv)	Takes different arguments than the default system version and requires that the <sys/types.h> header file is included. **Note** To access the reference page for the System V version of mount, make sure that the 2sv section specifier is included on the man command line.
open(2)	Specifies that the O_NOCTTY flag is not set by default as it is in the base system. Thus, if the proper conditions exist, an open call to a terminal device will allow the device to become the controlling terminal for the process.
pipe(2)	Supports a pipe operation on STREAMS-based file descriptors.
sigaction(2) and signal(2)	Specifies that the kernel pass additional information to the signal handler. This includes passing the reason that the signal was delivered (into the siginfo structure) and the context of the calling process when the signal was delivered into the ucontext structure.
sigpause(2)	Unblocks the specified signal from the calling process's signal mask and suspends the calling process until a signal is received. The SIGKILL and SIGSTOP signals cannot be reset.
sigset(2)	Specifies that if the disposition for SIGCHLD is set to SIG_IGN, the calling process's children cannot turn into zombies when they terminate. If the parent subsequently waits for its children, it blocks until all of its children terminate. This operation then returns a value of -1 and sets errno to [ECHILD].
unlink(2)	Does not allow users (including superusers) to unlink nonempty directories and sets errno to ENOTEMPTY. It allows superusers to unlink a directory if it is empty.

Table B–2: Library Function Summary

Library Functions	System V Behavior
`getcwd`(3)	Gets the name of the current directory: `char *`*`getcwd`*` (char * `*`buffer`*`, int `*`size`*` );`
`mkfifo`(3)	Supports creation of STREAMS-based FIFO and uses `/dev/streams/pipe`.
`mktemp`(3)	Uses the `getpid` function to obtain the *`pid`* part of the unique name.
`ttyname`(3)	Returns a pointer to a string with the pathname that begins with `/dev/pts/` when the terminal is a pseudoterminal device.

C

Creating Dynamically Configurable Kernel Subsystems

When you create a new kernel subsystem or modify an existing kernel subsystem, you can write the subsystem so that it is dynamically configurable. This appendix explains how to make a subsystem dynamically configurable by providing the following information:

- A conceptual description of a dynamically configurable subsystem (Section C.1)
- A conceptual description of the attribute table, including example attribute tables (Section C.2)
- An explanation of how to create a configuration routine, including an example configuration routine (Section C.3)
- A description of how to check the operating system version number to ensure that the subsystem is compatible with it (Section C.4)
- Instructions for building a loadable subsystem into the kernel for testing purposes (Section C.5)
- Instructions for building a static subsystem that allows run-time attribute modification into the kernel for testing purposes (Section C.6)
- Information about debugging a dynamically configurable subsystem (Section C.7)

Before the Tru64 UNIX system supported dynamically configurable subsystems, system administrators managed kernel subsystems by editing their system's configuration file. Each addition or removal of a subsystem or each change in a subsystem parameter required rebuilding the kernel, an often difficult and time-consuming process. System administrators responsible for a number of systems had to make changes to each system's configuration file and rebuild each kernel.

Dynamically configurable subsystems allow system administrators to modify system parameters, and load and unload subsystems without editing files and rebuilding the kernel. System administrators use the `sysconfig` command to configure the subsystems of their kernel. Using this command, system administrators can load and configure, unload and unconfigure, reconfigure (modify), and query subsystems on their local system and on remote systems.

Device driver writers should note device-driver specific issues when writing loadable device drivers. For information about writing loadable device drivers, see the *Writing Device Drivers* manual.

C.1 Overview of Dynamically Configurable Subsystems

Many Tru64 UNIX kernel subsystems are static, meaning that they are linked with the kernel at build time. After the kernel is built, these subsystems cannot be loaded or unloaded. An example of a static subsystem is the `vm` (virtual memory) subsystem. This subsystem must be present in the kernel for the system to operate correctly.

Some kernel subsystems are or can be loadable. A loadable subsystem is one that can be added to or removed from the kernel without rebuilding the kernel. An example of a subsystem that is loadable is the `presto` subsystem, which is loaded only when the Prestoserve software is in use.

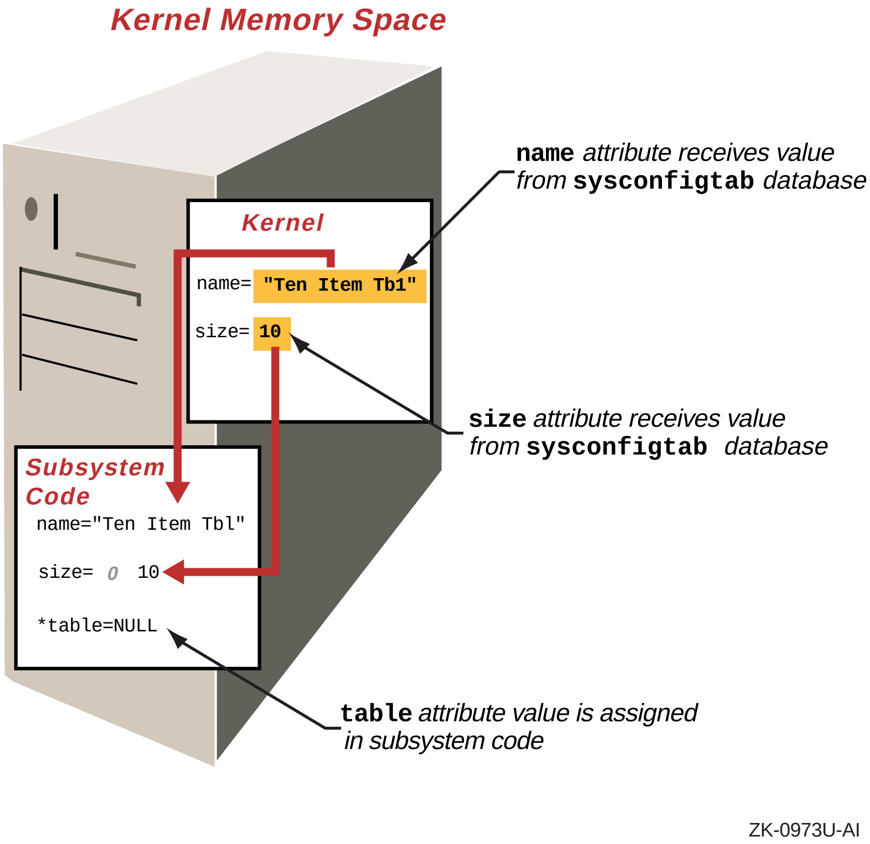
Both static and loadable subsystems can be dynamically configurable:

- For a static subsystem, dynamically configurable means that selected subsystem attributes can be modified without rebuilding the kernel. This type of subsystem can also answer queries about the values of its attributes and be unconfigured if it is not in use (however, it cannot be unloaded).
- For a loadable subsystem, dynamically configurable means that the subsystem is configured into the kernel at load time, can be modified without rebuilding the kernel, and is unconfigured before it is unloaded. This type of subsystem can also answer queries about its attributes.

Like traditional kernel subsystems, dynamically configurable subsystems have parameters. These parameters are referred to as *attributes*. Examples of subsystem attributes are timeout values, table sizes and locations in memory, and subsystem names. You define the attributes for the subsystem in an attribute table. (Attribute tables are described in Section C.2.)

Before initially configuring a loadable subsystem, system administrators can store values for attributes in the `sysconfigtab` database. This database is stored in the `/etc/sysconfigtab` file and is loaded into kernel memory at boot time. The values stored in this database become the initial value for the subsystem's attributes, whether your subsystem has supplied an initial value for the attribute. Figure C–1 demonstrates how initial attribute values come from the `sysconfigtab` database.

Figure C–1: System Attribute Value Initialization


Kernel Memory Space
Kernel
name= "Ten Item Tb1"
size= 10
Subsystem Code
name="Ten Item Tbl"
size= 0 10
*table=NULL
name attribute receives value from sysconfigtab database
size attribute receives value from sysconfigtab database
table attribute value is assigned in subsystem code
ZK-0973U-AI

Note that the `size` attribute in Figure C–1 receives its initial value from the `sysconfigtab` database even though the subsystem initializes the `size` attribute to 0 (zero).

Using an attribute table declared in the subsystem code, you control which of the subsystem's attribute values can be set at initial configuration. (For information about how you control the attributes that can be set in the `sysconfigtab` database, see Section C.2.)

In addition to being able to store attribute values for initial configuration, system administrators can query and reconfigure attribute values at any time when the subsystem is configured into the kernel. During a query request, attribute values are returned to the system administrator. During a reconfiguration request, attribute values are modified. How the return or modification occurs depends upon how attributes are declared in the subsystem code:

- If the subsystem's attribute table supplies the kernel with the address of an attribute, the kernel can modify or return the value of that attribute. Supplying an address to the kernel and letting the kernel handle the attribute value is the most efficient way to maintain an attribute value.
- If the kernel has no access to the attribute value, the subsystem must modify or return the attribute value. Although it is most efficient to let the kernel maintain attribute values, some cases require the subsystem to maintain the value. For example, the kernel cannot calculate the value of an attribute, so the subsystem must maintain values that need to be calculated.

Again, you control which of the subsystem's attribute values can be queried or reconfigured, as described in Section C.2.

In addition to an attribute table, each dynamically configurable subsystem contains a configuration routine. This routine performs tasks such as calculating the values of attributes that are maintained in the subsystem. This routine also performs subsystem-specific tasks, which might include, for example, determining how large a table needs to be or storing memory locations in local variables that can be used by the subsystem. (Section C.3 describes how you create the configuration routine.) The kernel calls the subsystem configuration routine each time the subsystem is configured, queried, reconfigured, or unconfigured.

Any subsystem that can be configured into the kernel can also be unconfigured from the kernel. When a system administrator unconfigures a subsystem from the kernel, the kernel memory occupied by that subsystem is freed if the subsystem is loadable. The kernel calls the subsystem configuration routine during an unconfigure request to allow the subsystem to perform any subsystem specific unconfiguration tasks. An example of a subsystem specific unconfiguration task is freeing memory allocated by the subsystem code.

C.2 Overview of Attribute Tables

The key to creating a good dynamically configurable subsystem is to declare a good attribute table. The attribute table defines the subsystem's attributes, which are similar to system parameters. (Examples of attributes are timeout values, table sizes and locations in memory, and so on.) The attribute table exists in two forms, the definition attribute table and the communication attribute table:

- The definition attribute table is included in your subsystem code. It defines the subsystem attributes. Each attribute definition is one element of the attribute table structure. The definitions include the name of the attribute, its data type, and a list of the requests that system administrators are allowed to make for that attribute. The definition of each attribute also includes its minimum and maximum values, and optionally its storage location. The kernel uses the attribute definition as it responds to configuration, reconfiguration, query, and unconfiguration requests from the system administrator.

- The communication attribute table is used for communication between the kernel and your subsystem code. Each element of this attribute table structure carries information about one attribute. The information includes the following:

  - The name and data type of the attribute
  - The request that has been made for an operation on that attribute
  - The status of the request
  - The value of the attribute

  This attribute table passes from the kernel to your subsystem each time the system administrator makes a configuration, reconfiguration, query, or unconfiguration request.

The reason for having two types of attribute tables is to save kernel memory. Some of the information in the definition attribute table and the communication attribute table (such as the name and data types of the attributes) is the same. However, much of the information differs. For example, the definition attribute table need not store the status of a request because no requests have been made at attribute definition time. Likewise, the communication attribute table does not need to contain a list of the supported requests for each attribute. To save kernel memory, each attribute table contains only the needed information.

Note

Attribute names defined in a subsystem attribute table must not begin with the string `method`. This string is reserved for naming attributes used in loadable device driver methods. For more information about device driver methods, see the *Writing Device Drivers* manual.

The following sections explain both types of attribute tables by showing and explaining their declaration in `/sys/include/sys/sysconfig.h`.

C.2.1 Definition Attribute Table

The definition attribute table has the data type `cfg_subsys_attr_t`, which is a structure of attributes declared as follows in the `/sys/include/sys/sysconfig.h` file:

```
typedef struct cfg_attr {
        char          name[CFG_ATTR_NAME_SZ]; 1
        uint          type; 2
        uint          operation; 3
        whatever      address; 4
        uint          min; 5
        uint          max;
        uint          binlength; 6
}cfg_subsys_attr_t;
```

1 The name of the attribute is stored in the `name` field. You choose this name, which can be any string of alphabetic characters, with a length between two and the value stored in the `CFG_ATTR_NAME_SZ` constant. The `CFG_ATTR_NAME_SZ` constant is defined in the `/sys/include/sys/sysconfig.h` file.

2 You specify the attribute data type in this field, which can be one of the data types listed in Table C–1.

Table C–1: Attribute Data Types

Data Type Name	Description
`CFG_ATTR_STRTYPE`	Null terminated array of characters (`char*`)
`CFG_ATTR_INTTYPE`	32-bit signed number (`int`)
`CFG_ATTR_UINTTYPE`	32-bit unsigned number (`unsigned`)
`CFG_ATTR_LONGTYPE`	64-bit signed number (`long`)
`CFG_ATTR_ULONGTYPE`	64-bit unsigned number
`CFG_ATTR_BINTYPE`	Array of bytes

3 The `operation` field specifies the requests that can be performed on the attribute. You specify one or more of the request codes listed in Table C–2 in this field.

The `CFG_OP_UNCONFIGURE` request code has no meaning for individual attributes because you cannot allow the unconfiguration of a single attribute.

Therefore, you cannot specify `CFG_OP_UNCONFIGURE` in the `operation` field.

Table C–2: Codes That Determine the Requests Allowed for an Attribute

Request Code	Meaning
`CFG_OP_CONFIGURE`	The value of the attribute can be set when the subsystem is initially configured.
`CFG_OP_QUERY`	The value of the attribute can be displayed at any time while the subsystem is configured.
`CFG_OP_RECONFIGURE`	The value of the attribute can be modified at any time while the subsystem is configured.

4 The `address` field determines whether the kernel has access to the value of the attribute.

If you specify an address in this field, the kernel can read and modify the value of the attribute. When the kernel receives a query request from the `sysconfig` command, it reads the value in the location you specify in this field and returns that value. For a configure or reconfigure request, the kernel checks for the following conditions:

- The data type of the new value is appropriate for the attribute.
- The value falls within the minimum and maximum values for the attribute.

If the value meets these requirements, the kernel stores the new value for the attribute. (You specify minimum and maximum values in the next two fields in the attribute definition.)

In some cases, you want or need to respond to query, configure, or reconfigure requests for an attribute in the subsystem code. In this case, specify a `NULL` in this field. For more information about how you control attribute values, see Section C.3.

5 The `min` and `max` fields define the minimum and maximum allowed values for the attribute. You choose these values for the attribute.

The kernel interprets the contents of these two fields differently, depending on the data type of the attribute. If the attribute is one of the integer data types, these fields contain minimum and maximum integer values. For attributes with the `CFG_ATTR_STRTYPE` data type, these fields contain the minimum and maximum lengths of the string. For attributes with the `CFG_ATTR_BINTYPE` data type, these fields contain the minimum and maximum numbers of bytes you can modify.

6 If you want the kernel to be able to read and modify the contents of a binary attribute, you use the `binlength` field to specify the current size of the binary data. If the kernel modifies the length of the binary data stored in the attribute, it also modifies the contents of this field.

This field is not used if the attribute is an integer or string or if you intend to respond to query and reconfigure request for a binary attribute in the configuration routine.

C.2.2 Example Definition Attribute Table

Example C–1 provides an example definition attribute table to help you understand its contents and use. The example attribute table is for a fictional kernel subsystem named `table_mgr`. The configuration routine for the fictional subsystem is shown and explained in Section C.3.

Example C–1: Example Attribute Table

```
#include <sys/sysconfig.h>
#include <sys/errno.h>

/*
 *   Initialize attributes
 */
static char                  name[] = "Default Table";
static int                   size = 0;
static long                  *table = NULL;


/*
 *  Declare attributes in an attribute table
 */

cfg_subsys_attr_t table_mgr_attrbutes[] = {
  /*
   * "name" is the name of the table
   */
  {"name",[1]                 CFG_ATTR_STRTYPE, [2]
   CFG_OP_CONFIGURE | CFG_OP_QUERY | CFG_OP_RECONFIGURE,[3]
   (caddr_t) name,[4]2, sizeof(name),[5]0[6]},
  /*
   * "size" indicates how large the table should be
   */
  {"size",                         CFG_ATTR_INTTYPE,
   CFG_OP_CONFIGURE | CFG_OP_QUERY | CFG_OP_RECONFIGURE,
   NULL, 1, 10, 0},
  /*
   * "table" is a binary representation of the table
   */
  {"table",                        CFG_ATTR_BINTYPE,
   CFG_OP_QUERY,
   NULL, 0, 0, 0},
  /*
   * "element" is a cell in the table array
```

Example C–1: Example Attribute Table (cont.)

```
   */
  {"element",                     CFG_ATTR_LONGTYPE,
   CFG_OP_QUERY | CFG_OP_RECONFIGURE,
   NULL, 0, 99, 0},
  {,0,0,0,0,0,0}   /* required last element */
};
```

The final entry in the table, `{,0,0,0,0,0,0}`, is an empty attribute. This attribute signals the end of the attribute table and is required in all attribute tables.

The first line in the attribute table defines the name of the table. This attribute table is named `table_mgr_attributes`. The following list explains the fields in the attribute `name`:

1. The name of the attribute is stored in the `name` field, which is initialized to `Default Table` by the data declaration that precedes the attribute table.

2. The attribute data type is `CFG_ATTR_STRTYPE`, which is a null-terminated array of characters.

3. This field specifies the operations that can be performed on the attribute. In this case, the attribute can be configured, queried, and reconfigured.

4. This field determines whether the kernel has access to the value of the attribute.

   If you specify an address in this field, as shown in the example, the kernel can read and modify the value of the attribute. When the kernel receives a query request from the `sysconfig` command, it reads the value in the location you specify in this field and returns that value. For a configure or reconfigure request, the kernel checks that the data type of the new value is appropriate for the attribute and that the value falls within the minimum and maximum values for the attribute. If the value meets these requirements, the kernel stores the new value for the attribute. (You specify minimum and maximum values in the next two fields in the attribute definition.)

5. These two fields define the minimum allowed value for the attribute (in this case, 2), and the maximum allowed value for the attribute (in this case, `sizeof(name)`).

   If you want the minimum and maximum values of the attribute to be set according to the system minimum and maximum values, you can use one of the constants defined in the `/usr/include/limits.h` file.

6 If you want the kernel to be able to read and modify the contents of a binary attribute, use this field to specify the current size of the binary data. If the kernel modifies the length of the binary data stored in the attribute, it also modifies the contents of this field.

This field is not used if the attribute is an integer or string or if you intend to respond to query and reconfigure request for a binary attribute in the configuration routine.

C.2.3 Communication Attribute Table

The communication attribute table, which is declared in the `/sys/include/sys/sysconfig.h` file, has the `cfg_attr_t` data type. As the following example shows, this data type is a structure of attributes:

```
typedef struct cfg_attr {
        char         name[CFG_ATTR_NAME_SZ]; 1
        uint         type; 2
        uint         status; 3
        uint         operation; 4
        long         index; 5
        union { 6
          struct {
                 caddr_t val;
                 ulong   min_len;
                 ulong   max_len;
                 void    (*disposal)();
               }str;
          struct {
                 caddr_t val;
                 ulong   min_size;
                 ulong   max_size;
                 void    (*disposal)();
                 ulong   val_size;
               }bin;
          struct {
                 caddr_t val;
                 ulong   min_len;
                 ulong   max_len;
               }num;
             }attr;
}cfg_attr_t;
```

1 The `name` field specifies the name of the attribute, following the same attribute name rules as the name field in the definition attribute table.

2 The `type` field specifies the data type of the attribute. Table C–1 lists the possible data types.

3 The `status` field contains a predefined status code. Table C–3 lists the possible status values.

Table C–3: Attribute Status Codes

Status Code	Meaning
CFG_ATTR_EEXISTS	Attribute does not exist.
CFG_ATTR_EINDEX	Invalid attribute index.
CFG_ATTR_ELARGE	Attribute value or size is too large.
CFG_ATTR_EMEM	No memory available for the attribute.
CFG_ATTR_EOP	Attribute does not support the requested operation.
CFG_ATTR_ESMALL	Attribute value or size is too small.
CFG_ATTR_ESUBSYS	The kernel is disallowed from configuring, responding to queries on, or reconfiguring the subsystem. The subsystem code must perform the operation.
CFG_ATTR_ETYPE	Invalid attribute type or mismatched attribute type.
CFG_ATTR_SUCCESS	Successful operation.

4 The `operation` field contains one of the operation codes listed in Table C–2.

5 The `index` field is an index into a structured attribute.

6 The `attr` union contains the value of the attribute and its maximum and minimum values.

For attributes with the `CFG_ATTR_STRTYPE` data type, the `val` variable contains string data. The minimum and maximum values are the minimum and maximum lengths of the string. The `disposal` routine is a routine you write to free the kernel memory when your application is finished with it..

For attributes with the `CFG_ATTR_BINTYPE` data type, the `val` field contains a binary value. The minimum and maximum values are the minimum and maximum numbers of bytes you can modify. The `disposal` routine is a routine you write to free the kernel memory when your application is finished with it. The `val_size` variable contains the current size of the binary data.

For numerical data types, the `val` variable contains an integer value and the minimum and maximum values are also integer values.

C.2.4 Example Communication Attribute Table

This section describes an example communication attribute table to help you understand its contents and use. The example attribute table is for a fictional kernel subsystem named `table_mgr`. The configuration routine for the fictional subsystem is shown and explained in Section C.3.

```
table_mgr_configure(
 cfg_op_t          op,              /*Operation code*/1
 caddr_t           indata,          /*Data passed to the subsystem*/2
 ulong             indata_size,     /*Size of indata*/
 caddr_t           outdata,         /*Data returned to kernel*/3
 ulong             outdata_size)    /*Count of return data items*/

{
```

The following list explains the fields in the `table_mgr_configure` communication attribute table:

1. The `op` variable contains the operation code, which can be one of the following:

   ```
   CFG_OP_CONFIGURE
   CFG_OP_QUERY
   CFG_OP_RECONFIGURE
   CFG_OP_UNCONFIGURE
   ```

2. The `indata` structure delivers data of `indata_size` to the configuration routine. If the operation code is `CFG_OP_CONFIGURE` or `CFG_OP_QUERY`, the data is a list of attribute names that are to be configured or queried. For the `CFG_OP_RECONFIGURE` operation code, the data consists of attribute names and values. No data is passed to the configuration routine when the operation code is `CFG_OP_UNCONFIGURE`.

3. The `outdata` structure and the `outdata_size` variables are placeholders for possible future expansion of the configurable subsystem capabilities.

C.3 Creating a Configuration Routine

To make the subsystem configurable, you must define a configuration routine. This routine works with the definition attribute table to configure, reconfigure, answer queries on, and unconfigure the subsystem.

Depending upon the needs of the subsystem, the configuration routine might be simple or complicated. Its purpose is to perform tasks that the kernel cannot perform for you. Because you can inform the kernel of the location of the attributes in the definition attribute table, it is possible for the kernel to handle all configure, reconfigure, and query requests for an attribute. However, the amount of processing done during these requests is then limited. For example, the kernel cannot calculate the value of an attribute for you, so attributes whose value must be calculated must be handled by a configuration routine.

The following sections describe an example configuration routine. The example routine is for a fictional `table_mgr` subsystem that manages a table of binary values in the kernel. The configuration routine performs these tasks:

- Allocates kernel memory for the table at initial configuration
- Handles queries about attributes of the table
- Modifies the size of the table when requested by the system administrator
- Frees kernel memory when unconfigured
- Returns to the kernel

Source code for this subsystem is included on the system in the `/usr/examples/cfgmgr` directory. The definition attribute table for this subsystem is shown in Section C.2.2. The communication attribute table for this subsystem is shown in Section C.2.4.

C.3.1 Performing Initial Configuration

At initial configuration, the `table_mgr` subsystem creates a table that it maintains. As shown in Example C–1, the system administrator can set the name and size of the table at initial configuration. To set these values, the system administrator stores the desired values in the `sysconfigtab` database.

The default name of the table, defined in the subsystem code, is `Default Table`. The default size of the table is zero elements.

The following example shows the code that is executed during the initial configuration of the `table_mgr` subsystem:

```
.
.
.
switch(op){ 1
 case CFG_OP_CONFIGURE:

   attributes = (cfg_attr_t*)indata; 2

   for (i=0; i<indata_size; i++){ 3
    if (attributes[i].status == CFG_ATTR_ESUBSYS) { 4

     if (!strcmp("size", attributes[i].name)){ 5
         /* Set the size of the table      */
         table = (long *) kalloc(attributes[i].attr.num.val*sizeof(long)); 6

         /*
          * Make sure that memory is available
          */

         if (table == NULL)  {  7
             attributes[i].status = CFG_ATTR_EMEM;
             continue;
         }

         /*
          * Success, so update the new table size and attribute status
          */
```

```
        size = attributes[i].attr.num.val; 8
        attributes[i].status = CFG_ATTR_SUCCESS;
        continue;
    }
   }
  }
break;
.
.
.
```

1. The configuration routine contains a `switch` statement to allow the subsystem to respond to the various possible operations. The subsystem performs different tasks, depending on the value of the `op` variable.

2. This statement initializes the pointer `attributes`. The configuration routine can now manipulate the data it was passed in the `indata` structure.

3. The `for` loop examines the status of each attribute passed to the configuration routine.

4. If the status field for the attribute contains the `CFG_ATTR_ESUBSYS` status, the configuration routine must configure that attribute.

5. For the initial configuration, the only attribute that needs to be manipulated is the `size` attribute. The code within the `if` statement is executed only when the `size` attribute is the current attribute.

6. When the status field contains `CFG_ATTR_ESUBSYS` and the attribute name field contains `size`, the local variable `table` receives the address of an area of kernel memory. The area of kernel memory must be large enough to store a table of the size specified in `attributes[i].attr.num.val`. The value specified in `attributes[i].attr.num.val` is an integer that specifies the number of longwords in the table. The kernel reads the integer value from the `sysconfigtab` database and passes it to the configuration routine in the `attr` union.

7. The `kalloc` routine returns `NULL` if it is unable to allocate kernel memory. If no memory has been allocated for the table, the configuration routine returns `CFG_ATTR_EMEM`, indicating that no memory was available. When this situation occurs, the kernel displays an error message. The subsystem is configured into the kernel, but the system administrator must use the `sysconfig` command to reset the size of the table.

8. If kernel memory is successfully allocated, the table size from the `sysconfigtab` file is stored in the static external variable `size`. The subsystem can now use that value for any operations that require the size of the table.

C.3.2 Responding to Query Requests

During a query request, a user of the `table_mgr` subsystem can request that the following be displayed:

- The name of the table
- The table size
- The table itself
- A single element of the table

As shown in Example C–1, the `name` attribute declaration includes an address (`(caddr_t) name`) that allows the kernel to access the name of the table directly. As a result, no code is needed in the configuration routine to respond to a query about the name of the table.

The following example shows the code that is executed as part of a query request:

```
switch (op):
.
.
.

  case CFG_OP_QUERY:
/*
 * indata is a list of attributes to be queried, and
 * indata_size is the count of attributes
 */
attributes = (cfg_attr_t *) indata;[1]

for (i = 0; i < indata_size; i++) {[2]
    if (attributes[i].status == CFG_ATTR_ESUBSYS) {[3]

 /*
  * We need to handle the query for the following
  * attributes.
  */

 if (!strcmp(attributes[i].name, "size")) {[4]

     /*
      * Fetch the size of the table.
      */
     attributes[i].attr.num.val = (long) size;
     attributes[i].status = CFG_ATTR_SUCCESS;
     continue;
 }

 if (!strcmp(attributes[i].name, "table")) {[5]

     /*
      * Fetch the address of the table, along with its size.
      */
     attributes[i].attr.bin.val = (caddr_t) table;
     attributes[i].attr.bin.val_size = size * sizeof(long);
     attributes[i].status = CFG_ATTR_SUCCESS;
     continue;
 }

 if (!strcmp(attributes[i].name, "element")) {[6]
```

```
     /*
      * Make sure that the index is in the right range.
      */
     if (attributes[i].index < 1 || attributes[i].index > size) {
         attributes[i].status = CFG_ATTR_EINDEX;
  continue;
     }

     /*
      * Fetch the element.
      */
     attributes[i].attr.num.val = table[attributes[i].index - 1];
     attributes[i].status = CFG_ATTR_SUCCESS;
     continue;
 }
    }
}

break;
.
.
.
```

1. This statement initializes the pointer `attributes`. The configuration routine can now manipulate the data that was passed to it in the `indata` structure.

2. The `for` loop examines the status of each attribute passed to the configuration routine.

3. If the status field for the attribute contains the `CFG_ATTR_ESUBSYS` status, the configuration routine must respond to the query request for that attribute.

4. When the current attribute is `size`, this routine copies the value stored in the `size` variable into the `val` field of the `attr` union (`attributes[i].attr.num.val`). Because the `size` variable is an integer, the `num` portion of the union is used.

   This routine then stores the status `CFG_ATTR_SUCCESS` in the status field `attributes[i].status`.

5. When the current attribute is `table`, this routine stores the address of the table in the `val` field of the `attr` union. Because this attribute is binary, the `bin` portion of the union is used and the size of the table is stored in the `val_size` field. The size of the table is calculated by multiplying the current table size, `size`, and the size of a longword.

   The status field is set to `CFG_ATTR_SUCCESS`, indicating that the operation was successful.

6. When the current attribute is `element`, this routine stores the value of an element in the table into the `val` field of the `attr` union. Each element is a longword, so the `num` portion of the `attr` union is used.

If the index specified on the sysconfig command line is out of range, the routine stores CFG_ATTR_EINDEX into the status field. When this situation occurs, the kernel displays an error message. The system administrator must retry the operation with a different index.

When the index is in range, the status field is set to CFG_ATTR_SUCCESS, indicating that the operation is successful.

C.3.3 Responding to Reconfigure Requests

A reconfiguration request modifies attributes of the table_mgr subsystem. The definition attribute table shown in Example C–1 allows the system administrator to reconfigure the following table_mgr attributes:

- The name of the table
- The size of the table
- The contents of one element of the table

As shown in Example C–1, the name attribute declaration includes an address ((caddr_t) name) that allows the kernel to access the name of the table directly. Thus, no code is needed in the configuration routine to respond to a reconfiguration request about the name of the table.

The following example shows the code that is executed during a reconfiguration request:

```
switch(op){
⋮

  case CFG_OP_RECONFIGURE:
     /*
      * The indata parameter is a list of attributes to be
      * reconfigured, and indata_size is the count of attributes.
      */
    attributes = (cfg_attr_t *) indata; 1

    for (i = 0; i < indata_size; i++) { 2
     if (attributes[i].status == CFG_ATTR_ESUBSYS) { 3

        /*
         * We need to handle the reconfigure for the following
         * attributes.
         */

         if (!strcmp(attributes[i].name, "size")) { 4

           long  *new_table;
           int   new_size;

           /*
            * Change the size of the table.
            */
            new_size = (int) attributes[i].attr.num.val; 5
            new_table = (long *) kalloc(new_size * sizeof(long));

            /*
```

```
             * Make sure that we were able to allocate memory.
             */
             if (new_table == NULL) {6
              attributes[i].status = CFG_ATTR_EMEM;
       continue;
      }

     /*
      * Update the new table with the contents of the old one,
      * then free the memory for the old table.
      */
      if (size) {7
        bcopy(table, new_table, sizeof(long) *
           ((size < new_size) ? size : new_size));
        kfree(table);
      }

     /*
      * Success, so update the new table address and size.
      */
      table = new_table;8
      size = new_size;
      attributes[i].status = CFG_ATTR_SUCCESS;
      continue;
  }

         if (!strcmp(attributes[i].name, "element")) {9

   /*
    * Make sure that the index is in the right range.
    */
    if (attributes[i].index < 1 || attributes[i].index > size) {10
        attributes[i].status = CFG_ATTR_EINDEX;
        continue;
    }

   /*
    * Update the element.
    */
    table[attributes[i].index - 1] = attributes[i].attr.num.val;11
    attributes[i].status = CFG_ATTR_SUCCESS;
    continue;
         }
}
     }

     break;

.
.
.
```

1 This statement initializes the pointer `attributes`. The configuration routine can now manipulate the data that was passed to it in the `indata` structure.

2 The `for` loop examines the status of each attribute passed to the configuration routine.

3 If the status field for the attribute contains the `CFG_ATTR_ESUBSYS` status, the configuration routine must reconfigure that attribute.

4 When the current attribute is size, the reconfiguration changes the size of the table. Because the subsystem must ensure that kernel memory is available and that no data in the existing table is lost, two new variables are declared. The new_table and new_size variables store the definition of the new table and new table size.

5 The new_size variable receives the new size, which is passed in the attributes[i].attr.num.val field. This value comes from the sysconfig command line.

The new_table variable receives an address that points to an area of memory that contains the appropriate number of bytes for the new table size. The new table size is calculated by multiplying the value of the new_size variable and the number of bytes in a longword (sizeof (long)).

6 The kalloc routine returns NULL if it was unable to allocate kernel memory. If no memory has been allocated for the table, the configuration routine returns CFG_ATTR_EMEM, indicating that no memory was available. When this situation occurs, the kernel displays an error message. The system administrator must reenter the sysconfig command with an appropriate value.

7 This if statement determines whether a table exists. If one does, then the subsystem copies data from the existing table into the new table. It then frees the memory that is occupied by the existing table.

8 Finally, after the subsystem is sure that kernel memory has been allocated and data in the existing table has been saved, it moves the address stored in new_table into table. It also moves the new table size from new_size into size.

The status field is set to CFG_ATTR_SUCCESS, indicating that the operation is successful.

9 When the current attribute is element, the subsystem stores a new table element into the table.

10 Before it stores the value, the routine checks to ensure that the index specified is within a valid range. If the index is out of the range, the routine stores CFG_ATTR_EINDEX in the status field. When this situation occurs, the kernel displays an error message. The system administrator must retry the operation with a different index.

11 When the index is in range, the subsystem stores the val field of the attr union into an element of the table. Each element is a longword, so the num portion of the attr union is used.

The status field is set to CFG_ATTR_SUCCESS indicating that the operation is successful.

C.3.4 Performing Subsystem-Defined Operations

The table_mgr subsystem defines an application-specific operation that doubles the value of all fields in the table.

When a subsystem defines its own operation, the operation code must be in the range of CFG_OP_SUBSYS_MIN and CFG_OP_SUBSYS_MAX, as defined in the <sys/sysconfig.h> file. When the kernel receives an operation code in this range, it immediately transfers control to the subsystem code. The kernel does no work for subsystem-defined operations.

When control transfers to the subsystem, it performs the operation, including manipulating any data passed in the request.

The following example shows the code that is executed in response to a request that has the CFG_OP_SUBSYS_MIN value:

```
  switch (op) {
⋮

    case CFG_OP_SUBSYS_MIN:

    /*
     * Double each element of the table.
     */
    for (i=0; ((table != NULL) && (i < size)); i++)
        table[i] *= 2;

    break;
⋮

  }
```

The code doubles the value of each element in the table.

C.3.5 Unconfiguring the Subsystem

When the table_mgr subsystem is unconfigured, it frees kernel memory. The following example shows the code that is executed in response to an unconfiguration request:

```
  switch(op){
⋮

   case CFG_OP_UNCONFIGURE:
    /*
     * Free up the table if we allocated one.
    */
    if (size)
        kfree(table, size*sizeof(long));
    size = 0;
    break;
    }

    return ESUCCESS;
}
```

This portion of the configuration routine determines whether memory has been allocated for a table. If it has, the routine frees the memory using `kfree` function.

C.3.6 Returning from the Configuration Routine

The following example shows the `return` statement for the configuration routine.

```
  switch(op){
⋮

   size = 0;
   break;
    }

    return ESUCCESS;
```

The subsystem configuration routine returns ESUCCESS on completing a configuration, query, reconfigure, or unconfigure request. The way this subsystem is designed, no configuration, query, reconfiguration, or unconfiguration request, as a whole, fails. As shown in the examples in Section C.3.1 and Section C.3.3, operations on individual attributes might fail.

In some cases, you might want the configuration, reconfiguration, or unconfiguration of a subsystem to fail. For example, if one or more key attributes failed to be configured, you might want the entire subsystem configuration to fail. The following example shows a return that has an error value:

```
  switch(op){
⋮

        if (table == NULL)  {
          attributes[i].status = CFG_ATTR_EMEM;
          return ENOMEM;       /*Return message from errno.h*/
        }
```

The `if` statement in the example tests whether memory has been allocated for the table. If no memory has been allocated for the table, the subsystem returns with an error status and the configuration of the subsystem fails.

The following messages, as defined in /sys/include/sys/sysconfig.h and /usr/include/errno.h files, are displayed:

```
No memory available for the attribute
Not enough core
```

The system administrator must then retry the subsystem configuration by reentering the sysconfig command.

Any nonzero return status is considered an error status on return from the subsystem. The following list describes what occurs for each type of request if the subsystem returns an error status:

- An error on return from initial configuration causes the subsystem to not be configured into the kernel.
- An error on return from a query request causes no data to be displayed.
- An error on return from an unconfiguration request causes the subsystem to remain configured into the kernel.

C.4 Allowing for Operating System Revisions in Loadable Subsystems

When you create a loadable subsystem, you should add code to the subsystem to check the operating system version number. This code ensures that the subsystem is not loaded into an operating system whose version is incompatible with the subsystem.

Operating system versions that are different in major ways from the last version are called major releases of the operating system. Changes made to the system at a major release can cause the subsystem to operate incorrectly, so you should test and update the subsystem at each major operating system release. Also, you might want to take advantage of new features added to the operating system at a major release.

Operating system versions that are different in minor ways from the last version are called minor releases of the operating system. In general, the subsystem should run unchanged on a new version of the operating system that is a minor release. However, you should still test the subsystem on the new version of the operating system. You might want to consider taking advantage of any new features provided by the new version.

To allow you to check the operating system version number, the Tru64 UNIX system provides the global kernel variables version_major and version_minor. The following example shows the code you use to test the operating system version:

```
⋮
extern int version_major;
extern int version_minor;

if (version_major != 5 && version_minor != 0)
    return EVERSION;
```

The code in this example ensures that the subsystem is running on the Version 5.0 release of the operating system.

C.5 Building and Loading Loadable Subsystems

After you have written a loadable subsystem, you must build it and configure it into the kernel for testing purposes. This section describes how to build and load a loadable subsystem. For information about how to build a static subsystem that allows run-time attribute configuration, see Section C.6.

The following procedure for building dynamically loadable subsystems assumes that you are building a subsystem named `table_mgr`, which is contained in the files `table_mgr.c` and `table_data.c`. To build this subsystem, follow these steps:

1. Move the subsystem source files into a directory in the `/usr/sys` area:

   ```
   # mkdir /usr/sys/mysubsys
   # cp table_mgr.c /usr/sys/mysubsys/table_mgr.c
   # cp table_data.c /usr/sys/mysubsys/table_data.c
   ```

   You can replace the `mysubsys` directory name with the directory name of your choice.

2. Edit the `/usr/sys/conf/files` file using the text editor of your choice and insert the following lines:

   ```
   #
   # table_mgr subsystem
   #
   MODULE/DYNAMIC/table_mgr        optional table_mgr Binary
   mysubsys/table_mgr.c            module table_mgr
   mysubsys/table_data.c           module table_mgr
   ```

   The entry in the `files` file describes the subsystem to the `config` program. The first line of the entry contains the following information:

   - The `MODULE/DYNAMIC/table_mgr` token indicates that the subsystem is a dynamic kernel module (group of objects) named `table_mgr`.
   - The `optional` keyword indicates that the subsystem is not required into the kernel.
   - The `table_mgr` identifier is the token that identifies the subsystem on the `sysconfig` and `autosysconfig` command lines. Use caution when choosing this name to ensure that it is unique with

respect to other subsystem names. You can list more than one name for the subsystem.

- The Binary keyword indicates that the subsystem has already been compiled and object files can be linked into the target kernel.

Succeeding lines of the files file entry give the pathname to the source files that compose each module.

3. Generate the makefile and related header files by entering the following command:

```
# /usr/sys/conf/sourceconfig BINARY
```

4. Change to the /usr/sys/BINARY directory and build the module as follows:

```
# cd /usr/sys/BINARY
# make table_mgr.mod
```

5. When the module builds without errors, move it into the /subsys directory so that the system can load it:

```
# cp table_mgr.mod /subsys/
```

6. Load the subsystem by using either the /sbin/sysconfig command or the /sbin/init.d/autosysconfig command.

   The following shows the command line you would use to load and configure the table_mgr subsystem:

```
# /sbin/sysconfig -c table_mgr
```

   If you want the subsystem to be configured into the kernel each time the system reboots, enter the following command:

```
# /sbin/init.d/autosysconfig add table_mgr
```

   The autosysconfig command adds the table_mgr subsystem to the list of subsystems that are automatically configured into the kernel.

C.6 Building a Static Configurable Subsystem Into the Kernel

After you have written a static subsystem that allows run-time attribute configuration, you must build it into the kernel for testing purposes. This section describes how to build a static subsystem that supports the dynamic configuration of attributes.

The following procedure for building dynamically loadable subsystems assumes that you are building a subsystem named table_mgr, which is contained in the file table_mgr.c:

1. Move the subsystem source files into a directory in the /usr/sys area:

```
# mkdir /usr/sys/mysubsys
# cp table_mgr.c /usr/sys/mysubsys/table_mgr.c
# cp table_data.c /usr/sys/mysubsys/table_data.c
```

You can replace the mysubsys directory name with the directory name of your choice.

2. Edit the /usr/sys/conf/files file using the text editor of your choice and insert the following lines:

```
#
# table_mgr subsystem
#
MODULE/STATIC/table_mgr         optional table_mgr Binary
mysubsys/table_mgr.c            module table_mgr
mysubsys/table_data.c           module table_mgr
```

   The entry in the files file describes the subsystem to the config program. The first line of the entry contains the following information:

   - The MODULE/STATIC/table_mgr token indicates that the subsystem is a static kernel module (group of objects) named table_mgr.
   - The optional keyword indicates that the subsystem is not required in the kernel.
   - The table_mgr identifier is the token that identifies the subsystem in the system configuration file. Use caution when choosing this name to ensure that it is unique with respect to other subsystem names. You can list more than one name for the subsystem.
   - The Binary keyword indicates that the subsystem has already been compiled and object files can be linked into the target kernel.

   Succeeding lines of the files file entry give the pathname to the source files that compose each module.

3. Rebuild the kernel by running the /usr/sbin/doconfig program:

```
# /usr/sbin/doconfig
```

4. Enter the name of the configuration file at the following prompt:

```
   *** KERNEL CONFIGURATION AND BUILD PROCEDURE ***
Enter a name for the kernel configuration file. [MYSYS]: MYSYS.TEST
```

   For purposes of testing the kernel subsystem, enter a new name for the configuration file, such as MYSYS.TEST. Giving the doconfig program a new configuration file name allows the existing configuration file to remain on the system. You can then use the existing configuration file to configure a system that omits the subsystem you are testing.

5. Select option 15 from the Kernel Option Selection menu. Option 15 indicates that you are adding no new kernel options.

6. Indicate that you want to edit the configuration file in response to the following prompt:

   ```
   Do you want to edit the configuration file? (y/n) [n] yes
   ```

   The doconfig program then starts the editor. (To control which editor is invoked by doconfig, define the EDITOR environment variable.) Add the identifier for the subsystem, in this case table_mgr, to the configuration file:

   ```
   options   TABLE_MGR
   ```

   After you exit from the editor, the doconfig program builds a new configuration file and a new kernel.

7. Copy the new kernel into the root (/) directory:

   ```
   # cp /usr/sys/MYSYS_TEST/vmunix /vmunix
   ```

8. Shut down and reboot the system:

   ```
   # shutdown -r now
   ```

Note

You can specify that the module is required in the kernel by replacing the optional keyword with the standard keyword. Using the standard keyword saves you from editing the system configuration file. The following files file entry is for a required kernel module, one that is built into the kernel regardless of its inclusion in the system configuration file:

```
#
# table_mgr subsystem
#
MODULE/STATIC/table_mgr         standard Binary
mysubsys/table_mgr.c            module table_mgr
mysubsys/table_data.c           module table_mgr
```

When you make an entry such as the preceding one in the files file, you add the subsystem to the kernel by entering the following doconfig command, on a system named MYSYS:

```
# /usr/sbin/doconfig -c MYSYS
```

Replace MYSYS with the name of the system configuration file in the preceding command.

This command builds a vmunix kernel that is described by the existing system configuration file, with the addition of the subsystem being tested, in this case, the table_mgr subsystem.

C.7 Testing Your Subsystem

You can use the sysconfig command to test configuration, reconfiguration, query, and unconfiguration requests on the configurable subsystem. When you are testing the subsystem, enter the sysconfig command with the –v option. This option causes the sysconfig command to display more information than it normally does. The command displays, on the /dev/console screen, information from the cfgmgr configuration management server and the kloadsrv kernel loading software. Information from kloadsrv is especially useful in determining the names of unresolved symbols that caused the load of a subsystem to fail.

In most cases, you can use dbx, kdebug, and kdbx to debug kernel subsystems just as you use them to test other kernel programs. If you are using the kdebug debugger through the dbx –remote command, the subsystem's .mod file must be in the same location on the system running dbx and the remote test system. The source code for the subsystem should be in the same location on the system running dbx. For more information about the setup required to use the kdebug debugger, see the *Kernel Debugging* manual.

If the subsystem is dynamically loadable and has not been loaded when you start dbx, you must enter the dbx addobj command to allow the debugger to determine the starting address of the subsystem. If the debugger does not have access to the starting address of the subsystem, you cannot use it to examine the subsystem data and set breakpoints in the subsystem code. The following procedure shows how to invoke the dbx debugger, configure the table_mgr.mod subsystem, and enter the addobj command:

1. Invoke the dbx debugger:

   ```
   # dbx -k /vmunix
   dbx version 3.11.4
   Type 'help' for help.

   stopped at [thread_block:1542 ,0xfffffc00002f5334]

   (dbx)
   ```

2. Enter the sysconfig command to initially configure the subsystem:

   ```
   # sysconfig -c table_mgr
   ```

3. Enter the addobj command as shown:

   ```
   (dbx) addobj /subsys/table_mgr.mod
   (dbx) p &table_mgr_configure
   0xffffffff895aa000
   ```

Be sure to specify the full pathname to the subsystem on the addobj command line. (If the subsystem is loaded before you begin the dbx session, you do not need to enter the addobj command.)

If you want to set a breakpoint in the portion of the subsystem code that initially configures the subsystem, you must enter the addobj command following the load of the subsystem, but before the kernel calls the configuration routine. To stop execution between the load of the subsystem and the call to its configuration routine, set a breakpoint in the special routine, subsys_preconfigure. The following procedure shows how to set this breakpoint:

1. Invoke the dbx debugger and set a breakpoint in the subsys_preconfigure routine, as follows:

   ```
   # dbx -remote /vmunix
   dbx version 3.11.4
   Type 'help' for help.

   stopped at [thread_block:1542 ,0xfffffc00002f5334]

   (dbx) stop in subsys_preconfigure
   (dbx) run
   ```

2. Enter the sysconfig command to initially configure the table_mgr subsystem:

   ```
   # sysconfig -c table_mgr
   ```

3. Enter the addobj command and set a breakpoint in the configuration routine:

   ```
   [5] stopped at   [subsys_preconfigure:1546
   ,0xfffffc0000273c58]
   (dbx) addobj /subsys/table_mgr.mod
   (dbx) stop in table_mgr_configure
   [6] stop in table_mgr_configure
   (dbx) continue
   [6] stopped at   [table_mgr_configure:47 ,0xffffffff895aa028]
   (dbx)
   ```

4. When execution stops in the subsys_preconfigure routine, you can use the dbx stack trace command, trace, to ensure that the configuration request is for the subsystem that you are testing. Then, set the breakpoint in the subsystem configuration routine.

D

Parallel Processing — Old Style

Parallel processing of Compaq C programs is supported in two forms on Tru64 UNIX systems:

- OpenMP interface — a parallel-processing interface defined by the OpenMP Architecture Review Board.
- Old-style parallel-processing interface — a parallel-processing interface developed prior to the OpenMP interface.

This appendix describes the old-style parallel-processing interface, that is, the language features supported before the OpenMP interface was implemented. See Chapter 13 for information about the OpenMP interface.

Note

> Programmers using the old-style interface should consider converting to the OpenMP interface, an industry standard.
>
> Anyone converting an application to parallel processing or developing a new parallel-processing application should use the OpenMP interface.

Understanding this appendix requires a basic understanding of the concepts of multiprocessing, such as what a thread is, if a data access is thread-safe, and so forth.

The parallel-processing directives use the `#pragma` preprocessing directive of ANSI C, which is the standard C mechanism for adding implementation-defined behaviors to the language. Because of this, the terms *parallel-processing directives* (or *parallel directives*) and *parallel-processing pragmas* are used somewhat interchangeably in this appendix.

This appendix contains information on the following topics:

- The general coding rules that apply to the use of parallel-processing pragmas (Section D.1)
- The syntax of the parallel-processing pragmas (Section D.2)
- Environment variables that can be used to control certain aspects of thread resource allocation at run time (Section D.3)

D.1 Use of Parallel-Processing Pragmas

This section describes the general coding rules that apply to all parallel-processing pragmas and provides an overview of how the pragmas are generally used.

D.1.1 General Coding Rules

In many ways, the coding rules for the parallel-processing pragmas follow the rules of most other pragmas in Compaq C. For example, macro substitution is not performed in the pragmas. In other ways, the parallel-processing pragmas are unlike any other pragmas in Compaq C. This is because while other pragmas generally perform such functions as setting a compiler state (for example, message or alignment), these pragmas are statements. For example:

- The pragmas must appear inside a function body.
- The pragmas affect the program execution (as described in this appendix).
- Most pragmas apply to the statement that follows them. If you wish the pragma to apply to more than one statement, you must use a compound statement (that is, the statements must be enclosed in curly braces).

Several of the pragmas can be followed by modifiers that specify additional information. To make using these modifiers easier, each one can appear on a separate line following the parallel-processing pragma as long as the line containing the modifiers also begins with `#pragma` followed by the modifier. For example:

```
#pragma parallel if(test_function()) local(var1, var2, var2)
```

This example can also be written as:

```
#pragma parallel
#pragma if(test_function())
#pragma local(var1, var2, var2)
```

Note that the modifiers themselves cannot be broken over several lines. For example, the earlier code could not be written as:

```
#pragma parallel
#pragma if(test_function()) local(var1,
#pragma var2, var2)
```

D.1.2 General Use

The `#pragma parallel` directive is used to begin a parallel region. The statement that follows the `#pragma parallel` directive delimits the extent of the parallel region. It is typically either a compound statement containing ordinary C statements (with or without other parallel-processing directives) or another parallel-processing directive (in which case the parallel region consists of that one statement). Within a compound statement delimiting a parallel region, any ordinary C statements not controlled by other parallel-processing directives simply execute on each thread. The C statements within the parallel region that are controlled by other parallel-processing directives execute according to the semantics of that directive.

All other parallel-processing pragmas, except for `#pragma critical`, must appear lexically inside a parallel region. The most common type of code that appears in a parallel region is a `for` loop to be executed by the threads. Such a `for` loop must be preceded by a `#pragma pfor`. This construct allows different iterations of the `for` loop to be executed by different threads, which speeds up the program execution. The following example shows the pragmas that might be used to execute a loop in parallel:

```
#pragma parallel local(a)
#pragma pfor iterate(a = 1 ; 1000 ; 1)
for(a = 0 ; a < 1000 ; a++) {
<loop code>
}
```

A loop that executes in parallel must obey certain properties. These include:

- The index variable is not modified by the loop except by the third expression of the `for` statement. Further, that expression must always adjust the index variable by the same amount.
- Each iteration of the loop must be independent. That is, the computations performed by one iteration of the loop must not depend on the results of another iteration
- The number of iterations of the loop is known before the loop starts.

The programmer is responsible for verifying that the parallel loops obey these restrictions.

Another use of parallel processing is to have several different blocks of code run in parallel. The `#pragma psection` and `#pragma sections` directives are used for this purpose. The following code shows how these directives might be used:

```
#pragma parallel
#pragma psection
{
   #pragma section
   {   <code block>
   }
   #pragma section
   {   <code block>
   }
   #pragma section
   {   <code block>
   }
}
```

Once again, certain restrictions apply to the code block. For example, one code block must not rely on computations performed in other code blocks.

The final type of code that can appear in a parallel region is serial code. Serial code is neither within a `#pragma pfor` nor a `#pragma psection`. In this case, the same code will be executed by all of the threads created to execute the parallel region. While this may seem wasteful, it is often desirable to place serial code between two `#pragma pfor` loops or `#pragma psection` blocks. Although the serial code will be executed by all of the threads, this construct is more efficient than closing the parallel region after the first `pfor` or `psection` and opening another before the second one. This is due to run-time overhead associated with creating and closing a parallel region.

Be careful when placing serial code in a parallel region. Note that the following statements could produce unexpected results:

```
a++;
b++
```

Unexpected results may occur because all threads will execute the statements, causing the variables `a` and `b` to be incremented some number of times. To avoid this problem, enclose the serial code in a `#pragma one processor` block. For example:

```
#pragma one processor
{
a++;
b++
}
```

Note that no thread can proceed past the code until it has been executed.

D.1.3 Nesting Parallel Directives

Nested parallel regions are not currently supported in Compaq C. If a parallel region lexically contains another parallel region, the compiler will issue an error. However, if a routine executing inside a parallel region calls another routine that then tries to enter a parallel region, this second parallel region will execute serially and no error will be reported.

With the exception of #pragma parallel, it is invalid for most parallel constructs to execute other parallel constructs. For example, when running the code in a #pragma pfor, #pragma one processor, #pragma section, or #pragma critical code block, the only other parallel-processing construct that can execute is a #pragma critical. In the case where one parallel-processing pragma is lexically nested within another, the compiler will issue an error for all illegal cases. However, if code running in a code block transfers to a routine that then executes one of these directives, the behavior is unpredictable.

(As noted earlier in this appendix, all parallel-processing pragmas, except for #pragma critical, must appear lexically inside a #pragma parallel region.)

D.2 Parallel-Processing Pragma Syntax

This section describes the syntax of each of the parallel-processing pragmas.

The following parallel-processing pragmas are supported by the old-style parallel-processing interface:

- #pragma parallel — Denotes a parallel region of code (Section D.2.1).
- #pragma pfor — Marks a for loop that is to be run in parallel (Section D.2.2).
- #pragma psection — Begins a number of code sections, each of which is to be run in parallel with the others (Section D.2.3).
- #pragma section — Specifies each code section within a psection area (Section D.2.3).
- #pragma critical — Protects access to a critical area of code so that only one thread at a time can execute it (Section D.2.4).
- #pragma one processor — A section of code that should be executed by only one thread (Section D.2.5).
- #pragma synchronize — Stops threads until they all reach this point (Section D.2.6).
- #pragma enter gate and #pragma exit gate — A more complex form of synchronization. No thread is allowed to leave the exit gate until all threads have passed the enter gate (Section D.2.7).

D.2.1 #pragma parallel

The `#pragma parallel` directive marks a parallel region of code. The syntax of this pragma is:

#pragma parallel [*parallel-modifiers...*] *statement-or-code_block*

The *`parallel-modifiers`* for `#pragma parallel` are:

```
local(variable-list)
byvalue(variable-list)
shared(variable-list)
if (expression) [[no]ifinline]
numthreads(numthreads-option)
```

`local`, `byvalue`, and `shared` modifiers

The *`variable-list`* argument to the `local`, `byvalue`, and `shared` modifiers is a comma-separated list of variables that have already been declared in the program. You can specify any number of `local`, `byvalue`, and `shared` modifiers. This is useful if one of the modifiers requires a large number of variables.

The variables following the `shared` and `byvalue` modifiers will be shared by each thread.

The variables following the `local` modifier will be unique to each thread. Note that the value of variables outside the region are not passed into the region. Inside the region, the value of variables on the `local` modifier is undefined. Putting a variable in the local list has the same effect as declaring that variable inside the parallel region.

These modifiers are provided only for compatibility with other C compilers. In Compaq C, all visible variables declared outside the parallel region can be accessed and modified within the parallel region, and are shared by all threads (unless the variable is specified in the `local` modifier). For example:

```
int a,b,c;
#pragma parallel local(a) shared(b) byvalue(c)
{
<code that references a, b, and c>
}
```

This is the same as:

```
int a,b,c;
#pragma parallel
{
int a;
<code that references a, b, and c>
}
```

`if` modifier

The expression following the `if` modifier specifies a condition that will determine whether the code in the parallel region will actually be executed in parallel by many threads or serially by a single thread. If the condition is nonzero, the code will be run in parallel. This modifier can be used to delay, until run time, the decision to parallelize or not.

Note that running a small amount of code in parallel may take more time than running the code serially. This is due to the overhead involved in creating and destroying the threads needed to run the code in parallel.

`noifinline` modifier

The `noifinline` modifier can only be used if the `if` modifier is present. The default value, `ifinline`, tells the compiler to generate two versions of the code within the parallel region: one to execute in parallel if the `if` expression is nonzero, and one to execute serially if the `if` expression is zero. The `noifinline` modifier tells the compiler to generate only one form of the code. The `noifinline` modifier will cause less code to be generated, but the code will execute more slowly for those cases in which the `if` expression is zero.

`numthreads` modifier

The *`numthreads-option`* is one of:

```
min=expr1, max=expr2
percent=expr
expr
```

In all cases, the expressions should evaluate to a positive integer value. The case of `numthreads(`*`expr`*`)` is equivalent to `numthreads(min=0,max=`*`expr`*`)`. If a `min` clause is specified, the code will be run in parallel only if *`expr1`* threads (or more) are available to execute the region. If a `max` clause is specified, the parallel region will be executed by no more than *`expr2`* threads. If a `percent` clause is specified, the parallel region will be executed by *`expr`* percent of the available threads.

An example of a parallel region is:

```
#pragma parallel local(a,b) if(func()) numthreads(x)
{
code
}
```

The region of code will be executed if `func` returns a nonzero value. If it is executed in parallel, at most `x` threads will be used. Inside the parallel region each thread will have a local copy of the variables `a` and `b`. All other variables will be shared by the threads.

D.2.2 #pragma pfor

The `#pragma pfor` directive marks a loop for parallel execution. A `#pragma pfor` can only appear lexically inside a parallel region. The syntax of this pragma is:

#pragma pfor iterate(*iterate-expressions*) [*pfor-options*] *for-statement*

As the syntax shows, the `#pragma pfor` must be followed by the `iterate` modifier. The *`iterate-expressions`* takes a form similar to a for loop:

```
index-variable = expr1 ; expr2 ; expr3
```

- The *`index-variable`* = *`expr1`* must match the first expression in the `for` statement that follows the `#pragma pfor`. To run correctly, the *`index-variable`* must be local to the parallel region.
- The *`expr2`* expression specifies the number of times the loop will execute.
- The *`expr3`* expression specifies the value to be added to the index variable during each iteration of the loop.

Note that the *`iterate-expressions`* are closely related to the expressions that appear in the `for` statement that follows the `#pragma pfor`. It is the programmer's responsibility to ensure that the information provided in the *`iterate-expressions`* correctly characterizes how the for loop will execute.

The *`pfor-options`* are:

```
schedtype(schedule-type)
chunksize(expr)
```

The `schedtype` option tells the run-time scheduler how to partition the iterations among the available threads. Valid *`schedule-type`* values are:

- `simple` — The scheduler will partition the iterations evenly among all of the available threads. This is the default.
- `dynamic` — The scheduler will give each thread the number of iterations specified by the `chunksize` expression.
- `interleave` — This is the same as `dynamic` except that the work is assigned to the threads in an interleaved way.

- `gss` — The scheduler will give each thread a varied number of iterations. This is like `dynamic`, but instead of giving each thread a fixed chunksize, the number of iterations will begin with a large number and end with a small number.

The `chunksize` option is required for a `schedtype` of either `dynamic` or `interleave`. It is used to specify the number of iterations.

D.2.3 #pragma psection and #pragma section

The `#pragma psection` and `#pragma section` directives designate sections of code that are to be executed in parallel with each other. These directives can only appear lexically inside a parallel region. The syntax of these pragmas is:

#pragma psection

{

#pragma section

stmt1

#pragma section

stmt2. . .

#pragma section

stmtn

}

These pragmas do not have modifiers. The `#pragma psection` must be followed by a code block enclosed in braces. The code block must consist only of `#pragma section` directives followed by a statement or a group of statements enclosed in braces. You can specify any number of `#pragma section` directives within a `psection` code block.

D.2.4 #pragma critical

The `#pragma critical` directive designates a section of code that is to be executed by no more than one thread at a time. The syntax of this pragma is:

#pragma critical [*lock-option*] *statement-or-code-block*

The *`lock-option`* can be one of:

- `block` — The lock is specific to this critical section. Other threads can execute other critical sections while this critical section is executing, but only one thread can execute this critical section. This option can only be specified for critical sections within a parallel region.

- `region` — The lock is specific to this parallel region. Other threads that are executing code lexically outside the parallel region can execute other critical sections, but no other critical section within the parallel region can execute. This option can only be specified for critical sections within a parallel region.
- `global` — The global lock. No other critical section can execute while this one is executing. This is the default value.
- *`expr`* — An expression that specifies a user-supplied lock variable. In this case, the expression must designate a 32-bit or 64-bit integer variable.

D.2.5 #pragma one processor

The `#pragma one processor` directive designates a section of code that is to be executed by only one thread. This directive can only appear inside a parallel region. The syntax of this pragma is:

#pragma one processor *statement-or-code-block*

D.2.6 #pragma synchronize

The `#pragma synchronize` directive prevents the next statement from being executed until all threads have reached this point. This directive can only appear inside a parallel region. The syntax of this pragma is:

#pragma synchronize

D.2.7 #pragma enter gate and #pragma exit gate

The `#pragma enter gate` and `#pragma exit gate` directives allow a more flexible form of synchronization than `#pragma synchronize`. These directives can only appear inside a parallel region. Each `#pragma enter gate` in the region must have a matching `#pragma exit gate`. The syntax of these pragmas are:

#pragma enter gate (*name*)

#pragma exit gate (*name*)

The *`name`* is an identifier that designates each gate. The names of gates are in their own name space; for example, a gate name of `foo` is distinct from a variable named `foo`. A gate name is not declared before it is used.

This type of synchronization operates as follows: No thread can execute the statement after the `#pragma exit gate` until all threads have passed the matching `#pragma enter gate`.

D.3 Environment Variables

Certain aspects of parallel code execution can be controlled by the values of environment variables in the process when the program is started. The environment variables currently examined at the start of the first parallel execution in the program are as follows:

- `MP_THREAD_COUNT` — Tells the run-time system how many threads to create. The default is to use the number of processors on the system as the number of threads to create.
- `MP_CHUNK_SIZE` — Tells the run-time system what chunksize to use if the user either asked for the `RUNTIME` schedule type or omitted the chunksize when asking for another schedule type that requires a chunksize.
- `MP_STACK_SIZE` — Tells the run-time system how many bytes of stack space to allocate for each thread when it creates threads. The default is quite small, and if you declare any large arrays as local, you need to specify stack that is large enough to allocate them in.
- `MP_SPIN_COUNT` — Tells the run-time system how many times to spin while waiting for a condition to become true.
- `MP_YIELD_COUNT` — Tells the run-time system how many times to alternate between calling `sched_yield` and testing the condition before really going to sleep by waiting for a Pthread condition variable.

You can set these environment variables to integer values by using the conventions of your command-line shell. If an environment variable is not set, the run-time system chooses a plausible default behavior (which is generally biased toward allocating resources to minimize elapsed time).

E

Handling Names of Device Special Files

The manner in which devices and device special files are named and organized was changed in Version 5.0 of the Tru64 UNIX operating system. For details on these changes, see dsfmgr(8) and the *System Administration* manual. Also, see the *Installation Guide* for information on the effects that various installation operations have on the accessibility of new- and old-style device special file names.

To support the changes that affect device handling, the operating system provides conversion routines that make it relatively easy to handle the transition between the new- and old-style names. This appendix documents those routines. These routines will continue to be provided until the old-style names are no longer supported by the system.

System components or applications that reference the names of device special files will need to modify their source code. The modifications can be done, as necessary, either by replacing all device special file names with their new names or by using the conversion routines to derive the new names.

The following rules apply to all of the conversion routines:

- Arguments will be returned if their pointers in the functions calls are non-null.
- Only the first argument is mandatory. However, the functions will not return useful information, except status information, unless other arguments are specified.

The individual routines — dsfcvt_btoc( ), dsfcvt_ctob( ), dsfcvt_ntoo( ), dsfcvt_oton( ), dsfcvt_noro( ), and dsfcvt_cdevtoname( ) — are described in the following list. Descriptions of the parameters follow the routine descriptions.

dsfcvt_btoc()

Convert block name to character name. The block device name is searched for in the device status database, and, if found, the corresponding character device name is then searched for. If it is found, the name is returned along with the hardware ID.

Synopsis:

```
int dsfcvt_btoc(
         const char *blk_name,
         char *chr_name,
         char *blk_path,
         char *chr_path,
         long *hardware_id);
```

Return values: ESUCCESS (both device names found), ENOENT (block device name not found), ENODEV (character device name not found).

dsfcvt_ctob()

Convert character name to block name. The character name is searched for in the device status database, and, if found, the corresponding block device is then searched for. If it is found, the name is returned along with the hardware ID.

Synopsis:

```
int dsfcvt_ctob(
         const char *chr_name,
         char *blk_name,
         char *chr_path,
         char *blk_path,
         long *hardware_id);
```

Return values: ESUCCESS (both device names found), ENOENT (character device name not found), ENODEV (block device name not found).

dsfcvt_ntoo()

Convert new name to old name. The new name is searched for in the device status database, and, if found, the old name, if any, is returned along with the hardware ID.

Synopsis:

```
int dsfcvt_ntoo(
         const char *new_name,
         char *old_name,
         char *new_path,
         char *old_path,
         long *hardware_id);
```

Return values: ESUCCESS (both device names found), ENOENT (new device name not found), ENODEV (old device name not found).

dsfcvt_oton()

Convert old name to new name. The old name is searched for in the device status database, and, if found, the new name is returned along with the hardware ID.

Synopsis:

```
int dsfcvt_oton(
         const char *old_name,
         char *new_name,
         char *old_path,
         char *new_path,
         long *hardware_id);
```

Return values: ESUCCESS (both device names found), ENOENT (old device name not found), ENODEV (new device name not found).

`dsfcvt_noro()`

Convert a name to old or new name. The input name (either new or old) is searched for in the device status database. The first old or new name that matches will be returned in the appropriate field, and the input name will be returned in its appropriate field. The hardware ID is also returned.

The search name may be in either or both name arguments. If in both, they must be the same name and must not point to the same string value.

Synopsis:

```
int dsfcvt_noro(
         char *new_name,
         char *old_name,
         char *new_path,
         char *old_path,
         long *hardware_id);
```

Return values: ESUCCESS (both device names found), ENOENT (input device name, old or new, not found), ENODEV (other device name, old or new, not found).

`dsfcvt_cdevtoname()`

Convert a device's unique numeric identifier (cdev) to new name and path. The `cdev` is searched for in the device status database, and, if found, the new name and path are returned.

Synopsis:

```
int dsfcvt_cdevtoname(
         dev_t cdev,
         char *new_name,
         char *new_path);
```

Return values: ESUCCESS (device found), ENOENT (device not found).

The following list provides information about the arguments to the functions:

blk_name | *chr_name*
Pointer to a name of a block or character device, respectively; for example, `dsk1a`, `rz4a`, or `rrz4a`.

blk_path | *chr_path*
Pointer to a path for a device identified by *blk_name* (for example, `/dev/disk/dsk1a`) or *chr_name* (for example, `/dev/rdisk/dsk1a`).

new_name | *old_name*
Pointer to a device name. The *new_name* argument follows the new-style naming conventions and the *old_name* argument follows the old-style naming conventions.

new_path | *old_path*
Pointer to a path for a device name. The *new_path* argument follows the new-style naming conventions and the *old_path* argument follows the old-style naming conventions — for example, `/dev/disk/dsk1a` and `/dev/rdisk/dsk1a` for block and character devices, respectively, under new-style conventions, or `/dev/rz4a` and `/dev/rrz4a` for block and character devices, respectively, under old-style conventions.

hardware_id
Pointer to where the unique numeric identifier for a device is to be returned. (This identifier is assigned and maintained by the kernel.)

cdev
The unique major-minor number assigned to a device by the kernel.

F

Optimizing Programs with -om and cord

This appendix describes how to optimize programs with the `cc` command's `-om` and `-cord` options. `-om` performs postlink optimization. `-cord` reorders the procedures in an executable program or shared library to improve cache utilization.

Note

The `spike` tool (see Section 10.1.3) is replacing the `cc` command's `-om` and `-cord` options, because it provides better control and more effective optimization. See also Chapter 10 and Chapter 8 for complete information about optimizing and profiling techniques, respectively.

This appendix is organized as follows:

- Using the `-om` postlink optimizer (Section F.1)
- Reordering procedures with `-cord` and feedback files (Section F.2)

F.1 Using the -om Postlink Optimizer

You can perform postlink optimizations by using the `cc` command's `-om` option. Section F.1.1 provide an overview of `-om`. Section F.1.2 shows how to use `-om` with feedback files for profile-directed optimization.

F.1.1 Overview

The `-om` postlink optimizer performs the following code optimizations:

- Removal of `nop` (no operation) instructions; that is, those instructions that have no effect on machine state.
- Removal of `.lita` data; that is, that portion of the data section of an executable image that holds address literals for 64-bit addressing. Using available options, you can remove unused `.lita` entries after optimization and then compress the `.lita` section.
- Reallocation of common symbols according to a size that you determine.

When you use the –om option by itself, you get the full range of postlink optimizations. You can request a particular optimization by specifying -om followed by one of the following options (for more information, see cc(1)):

```
-WL, -om_compress_lita
-WL, -om_dead_code
-WL, -om_feedback
-WL, -om_Gcommon,num
-WL, -om_ireorg_feedback,file
-WL, -om_no_inst_sched
-WL, -om_no_align_labels
-WL, -om_split_procedures
```

The -om option is most effective when used with the –non_shared option — that is, with executables. For example:

```
% cc -non_shared -O3 -om prog.c
```

The -om option should be specified in the final link operation.

F.1.2 Profile Directed Optimization with -om

The pixie profiler (see pixie(1)) provides profile information that the cc command's -om and -feedback options can use to tune the generated instruction sequences to the demands placed on the program by particular sets of input data. This technique is most effective with executables. For shared libraries,you can also use cord as described in Section F.2, or omit the -om option.

The following example shows the three necessary basic steps in this process, which consist of (1) preparing the program for profile-directed optimization, (2) creating an instrumented version of the program and running it to collect profiling statistics, and (3) feeding that information back to the compiler and linker to help them optimize the executable code. Later examples show how to elaborate on these steps to accommodate ongoing changes during development and data from multiple profiling runs.

```
% cc -feedback prog -o prog -non_shared -O3 *.c  1
% pixie -update prog 2
% cc -feedback prog -o prog -non_shared -om -O3 *.c  3
```

1 When the program is compiled with the -feedback option for the first time, a special augmented executable file is created. It contains information that the compiler uses to relate the executable to the source files. It also contains a section that is used later to store profiling feedback information for the compiler. This section remains empty after the first compilation, because the pixie profiler has not yet generated any feedback information (step 2). Make sure that the file name

specified with the -feedback option is the same as the executable file name, which in this example is prog (from the -o option). By default, the -feedback option applies the -g1 option, which provides optimum symbolization for profiling. You need to experiment with the -O*n* option to find the level of optimization that provides the best run-time performance for your program and compiler.

2 The pixie command creates an instrumented version of the program (prog.pixie) and then runs it (because a prof option, -update, is specified). Execution statistics and address mapping data are automatically collected in an instruction-counts file (prog.Counts) and an instruction-addresses file (prog.Addrs). The -update option puts this profiling information in the augmented executable.

3 In the second compilation with the -feedback option, the profiling information in the augmented executable guides the compiler and (through the -om option) the postlink optimizer. This customized feedback enhances any automatic optimization that the -O3 and -om options provide. You can make compiler optimizations even more effective by using the -ifo and/or -assume whole_program options in conjunction with the -feedback option. However, as noted in Section 10.1.1, the compiler may be unable to compile very large programs as if there were only one source file.

During a typical development process, steps 2 and 3 of the previous example are repeated as needed to reflect the impact of any changes to the source code. For example:

```
% cc -feedback prog -o prog -non_shared -O3 *.c
% pixie -update prog
% cc -feedback prog -o prog -non_shared -O3 *.c
[modify source code]
% cc -feedback prog -o prog -non_shared -O3 *.c
.....
[modify source code]
% cc -feedback prog -o prog -non_shared -O3 *.c
% pixie -update prog
% cc -feedback prog -o prog -non_shared -om -O3 *.c
```

Because the profiling information in the augmented executable persists from compilation to compilation, the pixie processing step that updates the information does not have to be repeated every time that a source module is modified and recompiled. But each modification reduces the relevance of the old feedback information to the actual code and degrades the potential quality of the optimization, depending on the exact modification. The pixie processing step after the last modification and recompilation guarantees that the feedback information is correctly updated for the last compilation.

The profiling information in an augmented executable file makes it larger than a normal executable (typically 3–5 percent). After development is completed, you can use the strip command to remove any profiling and symbol table information. For example:

```
% strip prog
```

You might want to run your instrumented program several times with different inputs to get an accurate picture of its profile. The following example explains how to merge profiling statistics from two runs of a program, prog, whose output varies from run to run with different sets of input:

```
% cc -feedback prog -o prog -non_shared -O3 *.c 1
% pixie -pids prog 2
% prog.pixie 3
(input set 1)
% prog.pixie
(input set 2)
% prof -pixie -update prog prog.Counts.* 4
% cc -feedback prog -o prog -non_shared -om -O3 *.c 5
```

1 The first compilation produces an augmented executable, as explained in the previous example.

2 By default, each run of the instrumented program (prog.pixie) produces a profiling data file called prog.Counts. The -pids option adds the process ID of each of the instrumented program's test runs to the name of the profiling data file that is produced (prog.Counts.*pid*). Thus, the data files that subsequent runs produce do not overwrite each other.

3 The instrumented program is run twice, producing a uniquely named data file each time — for example, prog.Counts.371 and prog.Counts.422.

4 The prof -pixie command merges the two data files. The -update option updates the executable, prog, with the combined information.

5 The second compilation step uses the combined profiling information from the two runs of the program to guide the optimization.

F.2 Profile Directed Reordering with -cord

The cc command's -cord option invokes the cord utility, which reorders the procedures in an executable program or shared library to improve instruction cache behavior. You use a feedback file that contains data from an actual run of your application as input to -cord; however, this is a different kind of feedback file (created with the pixie or prof -pixie command) than the kind discussed in Section F.1.2. The following example shows how to create a feedback file and then use the –cord option to compile an executable with the feedback file as input:

```
% cc -O3 -o prog *.c
% pixie -feedback prog.fb prog 1
% cc -O3 -cord -feedback prog.fb -o prog *.c  2
```

1. The pixie command creates an instrumented version of the program and also runs it (because a prof option, -feedback is specified). The -feedback option creates a feedback file (prog.fb) that collects execution statistics to be used by the compiler in the next step.
2. The cc command's -feedback option accepts the feedback file as input. The –cord option invokes the cord utility.

Compiling a shared library with feedback is similar. Profile the shared library with one or more programs that exercise the library code that most needs optimizing. For example:

```
% cc -o libexample.so -shared -g1 -O3 lib*.c 1
% cc -o exerciser -O3 exerciser.c -L. -lexample 2
% pixie -L. -incobj libexample.so -run exerciser 3
% prof -pixie -feedback libexample.fb libexample.so exerciser.Counts 4
% cc -cord -feedback libexample.fb -o libexample.so -shared -g1 -O3 lib*.c 5
```

1. The shared library is compiled with the -g1 option to give feedback data for each source line.
2. A program to exercise the important parts of the library is built.
3. The shared library and program are instrumented and run to profile them.
4. A feedback file is generated for just the shared library.
5. The shared library is recompiled, relinked, and reordered to optimize the performance of the code that the profile shows is most heavily used.

Use a feedback file generated with the same optimization level.

If you have produced a feedback file and plan to compile your program with the -non_shared option, it is better to use the feedback file with the -om option than with -cord.

You can also use cord with the runcord utility. For more information, see pixie(1), prof(1), cord(1), and runcord(1).

Index