# Digital UNIX

## Programmer’s Guide

Order Number: AA-PS30D-TE

March 1996

Product Version: Digital UNIX Version 4.0 or higher

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

**Digital Equipment Corporation**
**Maynard, Massachusetts**

Digital Equipment Corporation makes no representations that the use of its products in the manner described in this publication will not infringe on existing or future patent rights, nor do the descriptions contained in this publication imply the granting of licenses to make, use, or sell equipment or software in accordance with the description.

Possession, use, or copying of the software described in this publication is authorized only pursuant to a valid written license from Digital or an authorized sublicensor.

# 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

## A Using 32-Bit Pointers on Digital UNIX Systems

## B Differences in the System V Habitat

## C Dynamically Configurable Kernel Subsystems

## D Optimizing Techniques (MIPS-Based C Compiler)

## Index

## Examples

## Figures

## Tables

# About This Manual

This manual describes the programming environment of the Digital 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 Digital 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 to this manual for the Version 4.0 release of Digital UNIX:

- Chapter 2 – Removed information on pragmas from this chapter, creating a new Chapter 3 on pragmas.
- Chapter 7 – Created a new chapter documenting Third Degree, an Atom tool.
- Chapter 8 – Modified to include information on Atom tools used in profiling.
- Chapter 9 – Created a new chapter on using and developing Atom tools.
- Chapter 10 – Merged the contents of Chapter 4 from *System Tuning and Performance Management* into this chapter. Also, information on the `uopt` global optimizer (used with the `–oldc` version of the C compiler) has been moved to Appendix D.
- Chapter 12 – Modified to include information on TIS (Thread Independent Services) and the changes to `libc` functions to make them thread-safe.

## Organization

This manual contains twelve chapters and four appendixes.

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

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

Chapter 3 Describes the implementation-specific pragmas that are supported on the C compiler using the `–newc` and `–oldc` flags.

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. Includes information about the `dbx` commands, working with the monitor, setting breakpoints, and debuggging machine code.

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

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

Chapter 8 Describes how to use the `prof` and `gprof` tools to profile your code, enabling you to find which portions of code are consuming the most execution time.

Chapter 9 Discusses 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 post-link optimizer.

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

Chapter 12 Describes how to develop multithreaded programs.

Appendix A Describes how to use 32-bit pointers on 64-bit Digital UNIX systems.

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 global optimizer (`uopt`) used by the DEC OSF/1 C compiler (`–oldc`).

## 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*

*Digital Portable Mathematics Library*

*Writing Software for the International Market*

*Kernel Debugging*

*Ladebug Debugger Manual*

**Programming: Compatibility, Migration, and Standards**

*ULTRIX to Digital UNIX Migration Guide*

*VAX System V to Digital UNIX Migration Guide*

*System V Compatibility User’s Guide*

*POSIX Conformance Document*

*XPG3 Questionnaire*

**Programming: Realtime**

*Guide to Realtime Programming*

**Programming: Streams**

*Programmer’s Guide: STREAMS*

**Programming: Multithreaded Applications**

*Guide to DECthreads*

**General User Information**

*Release Notes*

*Documentation Overview*

The printed version of the Digital UNIX documentation set is color coded to help specific audiences quickly find the books that meet their needs. (You can order the printed documentation from Digital.) This color coding is reinforced with the use of an icon on the spines of books. The following list describes this convention:

| Audience | Icon | Color Code |
|---|---|---|
| General users | G | Blue |
| System and network administrators | S | Red |
| Programmers | P | Purple |
| Device driver writers | D | Orange |
| Reference page users | R | Green |

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

The *Documentation Overview, Glossary, and Master Index* provides information on all of the books in the Digital UNIX documentation set.

## Reader's Comments

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

You can send your comments in the following ways:

- Fax: 603-881-0120 Attn: UEG Publications, ZK03-3/Y32
- Internet electronic mail: `readers_comment@zk3.dec.com`

  A Reader's Comment form is located on line in the following location:

  `/usr/doc/readers_comment.txt`
- Mail:

  Digital Equipment Corporation<br>
  UEG Publications Manager<br>
  ZK03-3/Y32<br>
  110 Spit Brook Road<br>
  Nashua, NH 03062-9987

  A Reader's Comment form is located in the back of each printed manual. The form is postage paid if you mail it in the United States.

Please include the following information along with your comments:

- The full title of the book and the order number. (The order number is printed on the title page of this book and on its back cover.)

- The section numbers and page numbers of the information on which you are commenting.
- The version of Digital UNIX that you are using.
- If known, the type of processor that is running the Digital UNIX software.

The Digital 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 Digital technical support office. Information provided with the software media explains how to send problem reports to Digital.

## Conventions

`%`
`$` A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bourne and Korn shells.

`#` A number sign represents the superuser prompt.

`(dbx)` In examples, this symbol represents the `dbx` debugger's 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 ).

# Overview 1

This chapter describes phases in developing an application and which Digital UNIX tools to use during those phases. Topics in this chapter include the following:

- Specifications and design considerations
- Major software development tools
- Source file control
- Program installation tools
- Interprocess communications

## 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 Digital UNIX**

| Phase | Tools/Features |
|---|---|
| Requirements and specifications | Standards<br>Internationalization<br>Security |
| Design | Routines<br>Coding Considerations<br>Libraries<br>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 Digital 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. Digital 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 Digital 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 Digital 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

Adherence 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. For more information, see the *Application Environment Specification (AES) Operating System Programming Interfaces Volume*, ISBN 0-13-043522-8, published by Prentice-Hall, Inc.

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 Digital 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 Digital 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 Digital UNIX operating system provides interfaces and utilities for you to develop internationalized applications that conform to Issue 4 of 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 Digital UNIX internationalization package, see the manual *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*

*DECwindows Motif Guide to Application Programming*

*DECwindows Extensions to Motif*

*DECwindows Companion to the OSF/Motif Style Guide*

*Developing Applications for the Display PostScript System*

*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*

## 1.3 Major Software Development Tools

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

### 1.3.1 Languages That Run in the Digital UNIX Environment

The chief language that the Digital UNIX operating system supports is C. The Digital UNIX operating system includes a C language compiler. Other languages, such as Pascal and Fortran, are available separately. For a complete list of layered products, contact your Digital representitive. The Digital UNIX system also includes an assembler for working with assembly language. For more information on the assembler, see the `as`(1) reference page and the *Assembly Language Programmer's Guide*.

### 1.3.2 Linking Object Files

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

As part of the compilation process, most compilers call the linker (`ld`) to combine one or more object files into a single program object file. In addition, the linker resolves external references, searches libraries, and performs all other processing required to create object files that are ready for execution. The resulting object module can either be executed or serve as input for a separate `ld` run. (You can invoke the linker separately from the compiler by issuing the `ld` command.)

Digital UNIX allows you to create applications composed of source program modules written in different languages. In these instances, you compile each program module separately and then link the compiled modules together in a separate step.

Digital UNIX provides the ability to create shared libraries by using the `ld` command. In addition, you also can create archive (static) libraries by using the `ar` command. For more information, see Chapter 4. See Chapter 2 and Chapter 4, as well as the documentation sets for the individual languages, for detailed information on compiling and linking programs. For more information on the `ld` command, see the `ld`(1) reference page.

### 1.3.3 Debuggers

The following tools are the primary debugging tools on the Digital UNIX operating system:

- The `dbx` debugger (see Chapter 5 or `dbx`(1) for details)
- The Third Degree tool (see Chapter 7 or `third`(5) for details)
- The `lint` utility (see Chapter 6 or `lint`(1) for details)

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

The `dbx` debugger is the most comprehensive tool for debugging in a nonwindow environment.

## 1.4 Source File Control

An integral part of creating a software application is managing the development and maintenance processes. The Digital 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 Digital UNIX operating system provides support similar to SCCS and RCS utilities on other UNIX systems. In addition, Digital 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 the `sccs`(1) and `rcs`(1) reference pages and the manual *Programming Support Tools*.

## 1.5 Program Installation Tools

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

Software for Digital 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 Digital 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 manual *Programming Support Tools*.

## 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.

Digital UNIX provides the following facilities for interprocess communication:

- System V IPC

  System V IPC includes the following IPC facilities: messages, shared memory, and semaphores.

- Pipes

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

- 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 DECthreads* and Chapter 12.

- X/Open Transport Interface (XTI)

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

# The Compiler System 2

This chapter contains information on the following topics:

- Data types in the Digital UNIX environment
- Using the C preprocessor
- Linking object files
- Running programs
- Object file tools
- ANSI name space pollution cleanup in the standard C library

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. The output of this operation is an intermediate file with the `.i` file suffix.
- 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.

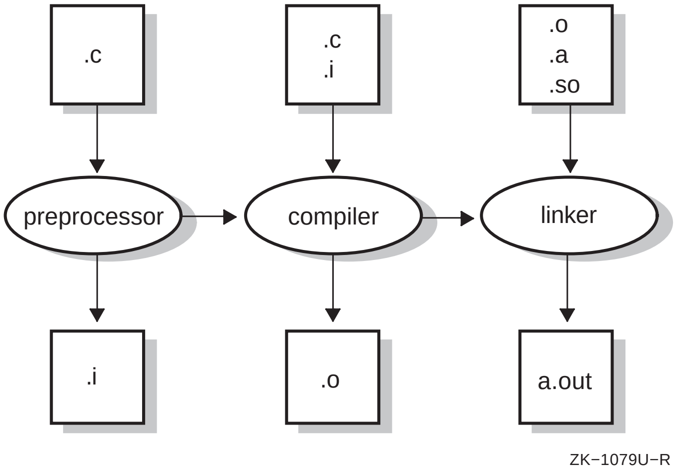
**Table 2-1: Compiler System Functions**

| 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*, *DEC C Language Reference Manual* |
| Debug programs | Symbolic debugger (`dbx` and `ladebug`) and Third Degree | Chapter 5, Chapter 6, `dbx`(1), `third`(5), `ladebug`(1), *Ladebug Debugger Manual* |
| Profile programs | Profiler, call graph profiler | Chapter 8, `prof`(1), `gprof`(1), `pixie`(5), `atom`(1), `hiprof`(5), `atomtools`(5) |
| Optimize programs | Optimizer, post-link optimizer | This chapter, Chapter 10, `cc`(1), `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 (Driver Programs)

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**



Compiler system commands, sometimes called driver programs, invoke the components of the compiler system. Each language has its own set of compiler commands and flags. In addition, your system might include layered products such as C++, or other languages such as Fortran or Pascal. The languages supported by any one system are determined by the choices made at the time the system is installed or modified. Thus, the configuration of your particular system may not support languages other than C and assembly.

The `cc` command invokes the C compiler. The `−newc` and `−oldc` flags invoke different compiler implementations (where the implementation invoked by `−newc` is upwardly compatible with that invoked by `−oldc`). The `−newc` compiler offers improved optimization, additional features, and greater compatibility with Digital compilers provided on other platforms. The `−newc` compiler implementation is the default.

The `−newc` compiler was accessible in previous versions of the Digital UNIX operating system by means of the `−migrate` flag. The `−newc` compiler has been made more compatible with the `−oldc` compiler.

### Note

> This manual uses the phrase ‘‘the C compiler’’ to refer to both versions of the DEC C compiler, `−newc` and `−oldc`. Features supported by only one of the compilers are so marked.

Each compiler implementation supports a slightly different set of compiler flags. See Table 2-4 for a comparison.

In the Digital UNIX programming environment, a single 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.
- Unlike the compilers, the assembler (`as`) can assemble only a single file, which is assumed to contain assembler code (any file suffix is ignored). The `as` command does not automatically link the assembled object file. Thus, if you directly invoke the assembler, you need to link the object in a separate step.
- Prevent linking and the creation of the executable program, thereby retaining the `.o` object file for a subsequent link operation.
- Pass the major flags associated with the link command (`ld`) to the linker. For example, you can include the `–L` flag 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) |

**Table 2-2: (continued)**

| Suffix | File |
|---|---|
| `.u` | ucode object file (supported only under `–oldc`) |
| `.b` | ucode object library (supported only under `–oldc`) |

## 2.2 Data Types in the Digital UNIX Environment

The following sections describe how data is represented on the Digital UNIX system.

### 2.2.1 Data Type Sizes

The Digital UNIX system is little endian; that is, the address of a multibyte integer is the address of its least significant byte; 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 |

### 2.2.2 Floating-Point Range and Processing

The C compiler supports IEEE single-precision (32-bit `float`) and double-precision (64-bit `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

Digital 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 nonnumbers [NaNs]) by specifying a compilation flag, 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 flags 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 additional information on how to handle IEEE floating-point exceptions.

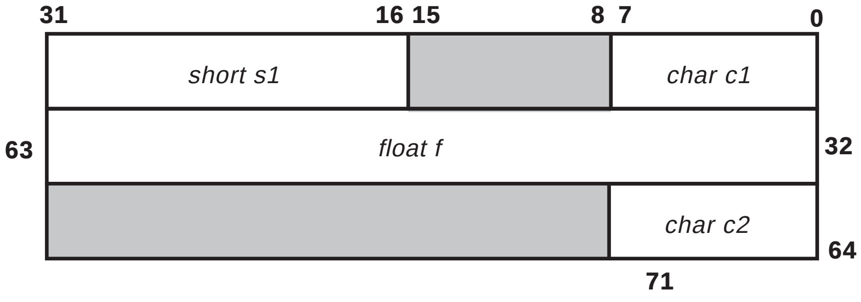
### 2.2.3 Structure Alignment

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−R

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 alignment, three bytes of padding are added after c2.

The following mechanisms can be used to override the default alignment of structure members:

- The #pragma member_alignment and #pragma nomember_alignment directives (–newc only)
- The #pragma pack directive (–newc or –oldc)
- The –Zp*n* flag

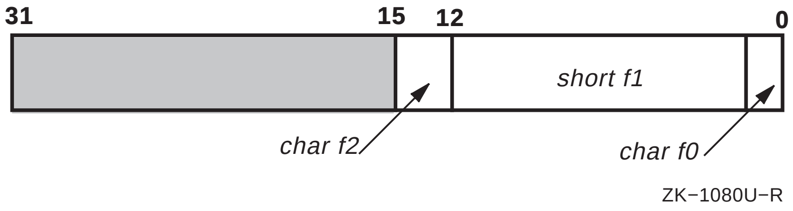
See Section 3.5 and Section 3.7 for information on these directives.

### 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 (–newc or –oldc) or the –Zp*n*compiler flag.) 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 flag 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.

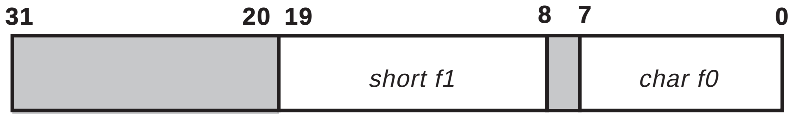
Similarly, if the –Zp2 flag or the #pragma pack 2 directive were used, the unnamed bit field would start at offset 16. With –Zp4 or #pragma pack 4, it would start at offset 32.

- 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 flag or the

**Figure 2-4: Padding to the Next Pack Boundary**



ZK−1081U−R

### 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, supported only by the C compiler using the –std and –newc flags (the default), 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 Digital UNIX -specific operations performed by the C preprocessor. For more information on the C preprocessor, see the `cc`(1) and `cpp`(1) reference pages and the *DEC 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 may run. You can reference these macros in `#ifdef` statements to isolate code that applies to a particular language or environment. The preprocessor macros are listed in Table 2-3.

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` flag enforces the ANSI C standard, but allows some common programming practices disallowed by the standard, and passes the macro `__STDC__=0` to the preprocessor.
- The `–std0` flag enforces the 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 causes the `__STDC__` macro to be undefined.
- The `–std1` flag strictly enforces the ANSI C standard and all 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 passes the macro `__STDC__=1` to the preprocessor.

### Table 2-3: Predefined Macros

| Macro | Source File Type | –std Flag |
|---|---|---|
| __DECC (–newc only) | .c | –std0, –std, –std1 |
| LANGUAGE_C | .c | –std0 |
| __LANGUAGE_C__ | .c | –std0, –std, –std1 |
| unix | .c, .s | –std0 |
| __unix__ | .c, .s | –std0, –std, –std1 |
| __osf__ | .c, .s | –std0, –std, –std1 |
| __alpha | .c, .s | –std0, –std, –std1 |
| SYSTYPE_BSD | .c, .s | –std0 |
| _SYSTYPE_BSD | .c, .s | –std0, –std, –std1 |
| LANGUAGE_ASSEMBLY | .s | –std0, –std, –std1 |
| __LANGUAGE_ASSEMBLY__ | .s | –std0, –std, –std1 |

### 2.3.2 Including Common Files

When writing programs, you often use header files that are common among a program's modules. These files define constants, the parameters for system calls, and so on.

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***"**

Enter this line in column 1 of a source file to indicate 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` flag, and finally in `/usr/include`.

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

Enter this line in column 1 of a source file to indicate that the C macro preprocessor should search for the include file `filename` only in the search path indicated by the `–I` flag and in `/usr/include`, but not in the current directory.

You can also use the `–Idir` compiler flag to specify additional pathnames (directories) to be searched by the C preprocessor for `#include` files. The C preprocessor searches first in the directory where the source file resides, followed by the specified pathname, *dir*, then the default directory, `/usr/include`. If *dir* is omitted, the default directory, `/usr/include`, is not searched.

### 2.3.3 Setting Up Multilanguage Include Files

C, Fortran, and assembly code can reside in the same include files, then 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 environment`
- `#pragma function`
- `#pragma inline`
- `#pragma intrinsic`
- `#pragma linkage`
- `#pragma member`
- `#pragma message`
- `#pragma pack`
- `#pragma pointer_size`
- `#pragma use_linkage`
- `#pragma weak`

The pragmas are described in detail in Chapter 3.

## 2.4 Compiling Source Programs

The `cc` command provides more than one compilation environment: The `-newc` and `-oldc` flags invoke different compiler implementations (where the implementation invoked by –newc is upwardly compatible with that invoked by –oldc). The –newc compiler offers improved optimization, additional features, and greater compatibility with Digital compilers provided on other platforms. The –newc compiler implementation is the default.

The –newc compiler has been accessible in previous versions of the Digital UNIX operating system by means of the –migrate flag. The –newc compiler has been made more compatible with the –oldc compiler.

All compilation environments produce object files that comply with the common object file format (COFF), and their objects files can be freely intermixed. The C compiler invoked by the –oldc flag employs ucode-based optimizations; the C compiler invoked by the –newc flag employs other optimizations.

The following sections describe the flags that are available in all compilation environments, the default compiler behavior, and how to compile multilanguage programs.

### 2.4.1 Compilation Flags

Compiler flags select a variety of program development functions, including debugging, optimizing, and profiling facilities, and the names assigned to output files.

Table 2-4 compares the flags that are available with the three compilation environments. An asterisk (*) indicates that the flag is accepted, but ignored, by the compiler. See the cc(1) reference page for more information on these flags.

**Table 2-4: Comparison of Compiler Flags**

| Flag | -newc | -oldc | -migrate |
|---|---|---|---|
| –ansi_alias | yes | no | yes |
| –[no_]ansi_args | yes | no | yes |
| –assume [no]accuracy_sensitive | yes | yes | yes |
| –assume [no]aligned_object | yes | no | yes |
| –assume [no]trusted_short_alignment | yes | no | yes |
| –B | yes | yes | yes |
| –c | yes | yes | yes |
| –C | yes | yes | yes |
| –call_shared | yes | yes | yes |
| –check | yes | no | yes |

**Table 2-4: (continued)**

| Flag | -newc | -oldc | -migrate |
|---|---|---|---|
| –compress | yes | yes | yes |
| –cord | yes | yes | yes |
| –[no_]cpp | yes | yes | yes |
| –D | yes | yes | yes |
| –double | yes | yes | yes |
| –edit | yes | yes | yes |
| –exact_version | yes | yes | yes |
| –E | yes | yes | yes |
| –fast | yes | yes | yes |
| –feedback | yes | yes | yes |
| –float | yes | yes | yes |
| –float_const | yes | yes | yes |
| –[no_]fp_reorder | yes | yes | yes |
| –fprm {c \| d \| n \| m} | yes | yes | yes |
| –fptm {n \| su \| sui \| u} | yes | yes | yes |
| –framepointer | yes | yes | yes |
| –g | yes | yes | yes |
| –G | yes* | yes | yes* |
| –gen_feedback | yes | no | yes |
| –h | yes | yes | yes |
| –H | yes | yes | yes |
| –I | yes | yes | yes |
| –ieee | yes | yes | yes |
| –ifo | yes | yes* | yes |
| –inline | yes | no | yes |
| –j | no | yes | no |
| –k | yes | yes | yes |
| –K | yes | yes | yes |
| –ko | yes | yes | yes |
| –M | yes | yes | yes |
| –machine_code | yes | no | yes |
| –MD | yes | yes | yes |

### Table 2-4: (continued)

| Flag | -newc | -oldc | -migrate |
|---|---|---|---|
| –[no_]misalign | yes | yes | yes |
| –no_archive | yes | yes | yes |
| –no_inline | yes | yes | yes |
| –nomember_alignment | yes | no | yes |
| –non_shared | yes | yes | yes |
| –noobject | yes | no | yes |
| –o | yes | yes | yes |
| –O | yes | yes | yes |
| –oldcomment | yes | yes | yes |
| –Olimit | yes* | yes | yes* |
| –p | yes | yes | yes |
| –P | yes | yes | yes |
| –[no_]pg | yes | yes | yes |
| –portable | yes | no | yes |
| –preempt_module | yes | no | yes |
| –preempt_symbol | yes | no | yes |
| –proto[is] | yes | yes | yes |
| –pthread | yes | yes | yes |
| –Q | yes | yes | yes |
| –readonly_strings | yes | yes | yes |
| –resumption_safe | yes | yes | yes |
| –S | yes | yes | yes |
| –scope_safe | yes | yes | yes |
| –show | yes | no | yes |
| –signed | yes | yes | yes |
| –source_listing | yes | no | yes |
| –speculate | yes | no | yes |
| –std[n] | yes | yes | yes |
| –t | yes | yes | yes |
| –taso | yes | yes | yes |
| –threads | yes | yes | yes |
| –tune | yes | yes | yes |

**Table 2-4: (continued)**

| Flag | -newc | -oldc | -migrate |
|---|---|---|---|
| –traditional | yes | yes | yes |
| –trapuv | yes | yes | yes |
| –U | yes | yes | yes |
| –unroll | yes | no | yes |
| –unsigned | yes | yes | yes |
| –v | yes | yes | yes |
| –V | yes | yes | yes |
| –varargs | yes | yes | yes |
| –vaxc | yes | no | yes |
| –verbose | yes | yes | yes |
| –volatile | yes | yes | yes |
| –w | yes | yes[a] | yes |
| –W | yes | yes | yes |
| –warnprotos | yes | yes | yes |
| –writable_strings | yes | yes | yes |
| –xtaso | yes | yes | yes |
| –xtaso_short | yes | yes | yes |
| –Zp | yes | yes | yes |

Table note:

a. The `–w0` flag is not accepted by the `–oldc` flag.

### 2.4.2 Default Compilation Behavior

Some flags have default values that are used if the flag 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 modules prog1.o and prog2.o and the executable program a.out.

Whether you are new to Digital UNIX, porting applications from other systems, or concerned with compatibility issues, knowing the default behavior of the compiler is useful. When you enter the cc compiler command with no other flags, the following flags are in effect:

–newc
: The default compiler flag; invoked when the compiler flag is not specified.

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

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

–double
: Promotes expressions of type float to double.

–fprm n
: Performs normal rounding (unbiased round to nearest) of floating-point numbers.

–g0
: Does not produce symbol information for symbolic debugging.

–I/usr/include
: Specifies that #include files whose names do not begin with / 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.

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

–no_fp_reorder
: Directs the compiler to use only certain scalar rules for calculations.

–no_misalign
: Generates alignment faults for arbitrarily aligned addresses.

–O1

Enables global optimizations.

`–oldcomment`
Allows traditional token concatenation.

`–p0`
Disables profiling.

`–no_pg`
Turns off `gprof` profiling.

`–preempt_symbol`
Allows symbol preemption on a symbol-by-symbol basis.

`–signed`
Causes all `char` declarations to be `signed char`.

`–std0`
Enforces the K&R standard with some ANSI extensions.

`–tune generic`
Selects instruction tuning that is appropriate for all implementations of the Alpha architecture.

`–unroll 0`
Directs the optimizer to use its own default loop unrolling amount.

`–writeable_strings`
Makes string literals writable.

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

- The output file is named `a.out` unless another name is specified by using the `–o` flag.
- Source files are linked automatically if compilation (or assembly) is successful.
- 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.

### 2.4.3 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` flag, 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`.

Most language driver programs pass information to `cc`, which, after processing, passes information to `ld`. When one of the modules to be compiled is a C program, you can usually use the driver command of the other language to compile and link both modules.

## 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 Using 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` flag 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` flag:

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

The `–lm` flag specifies the math library; the `–lexc` flag specifies the exception library.

You should 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 flags. For example:

- The –O0 flag requests no optimization (usually for debugging purposes).
- The –O1 flag requests certain local (module-specific) optimizations.
- Cross-module optimizations can be requested with the –O3 flag to the C compiler using the –oldc flag, or with the –ifo flag to the C compiler using the –newc flag. In this case, compiling multiple files in one operation allows the compiler to perform the maximum possible optimizations.
- Certain compilers may provide a combination of flags (such as –c and –o) that compile multiple source files into a single object module. This combination allows interprocedural optimizations to occur, yet retains the object file.

### 2.5.2 Linking Using the ld Command

Normally, users 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 flags and libraries that affect the linking process, see the ld(1) reference page.

### 2.5.3 Specifying Libraries

When you compile your program on the Digital 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 flags, 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 (non-shared 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 for the appropriate language compiler driver.

- **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 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.

- **When storing ucode object libraries**

  To link from a ucode library, specify the `–kl`*`x`* compiler flag.

### Note

Only the `–oldc` flag to the C compiler can be used to produce ucode files.

The following example links a file from a ucode library:

```
% cc -klucode_lib -o output main.u more.u rest.u
```

Because the libraries are searched as they are encountered on the command line, the order in which you specify them is important. Although a library might be made from both assembly and high-level language routines, the ucode object library contains code only for the high-level language routines.

Unlike an extended COFF object library, the ucode library does not contain code for the routines. You must specify to the ucode linker both the ucode object library and the extended COFF object library, in that order, to ensure that all modules are linked with the proper library.

If the compiler driver is to perform both a ucode link step and a final link step, the object file created after the ucode link step is placed in the position of the first ucode file specified or created on the command line in the final link step.

## 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 tables 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 the `strings`(1) reference page 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 ValueSclass    Symtype   Ref
echo-args.o:
[0]   main0x0000000000000000  0x01 0x06 0xfffff
[1]   printf   0x0000000000000000  0x06 0x06 0xfffff
[2]   _fpdata  0x0000000000000000  0x06 0x01 0xfffff
```

For more information, see the `odump`(1) reference page.

### 2.7.2 Listing Symbol Table Information (nm)

The `nm` tool displays symbol table information for object files. For example, `nm` would display 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 a debugging flag, 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)
  .
  .
  .
```

## 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 `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 a capital 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 preempt 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 Digital UNIX Version 4.0 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 preempt them using the nonreserved names. This kind of preemption is not ANSI compliant and is highly discouraged. However, the ability to preempt 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.

# Pragma Preprocessor Directives 3

This chapter describes the implementation-specific pragmas that are supported on the C compiler:

- `#pragma environment` (Section 3.1)
- `#pragma inline` (Section 3.2)
- `#pragma intrinsic` and `#pragma function` (Section 3.3)
- `#pragma linkage` (Section 3.4)
- `#pragma member_alignment` (Section 3.5)
- `#pragma message` (Section 3.6)
- `#pragma pack` (Section 3.7)
- `#pragma pointer_size` (Section 3.8)
- `#pragma use_linkage` (Section 3.9)
- `#pragma weak` (Section 3.10)

All of these pragmas can be used with the `–newc` or `–migrate` flags. A subset of these pragmas can be used with the `–oldc` flag:

```
#pragma function
#pragma intrinsic
#pragma pack
#pragma pointer_size
#pragma weak
```

## 3.1 The #pragma environment Directive

The C compiler (using the `–newc` flag) supports a `#pragma environment` directive that allows you to set, save, and restore the state of all context pragmas. The context pragmas are:

```
#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 like 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` and 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.

## 3.2 The #pragma inline Directive

Function inlining (supported by the C compiler using the `–newc` flag) 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 C compiler (using the `–newc` flag) enables the following preprocessor directives to 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. The definition can appear either before or after the function call.

The `cc` command flags `–O4` (for `–oldc`), `–O3` and `–O4` (for `–newc`), `-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 (using the `–newc` flag) inlines small static routines only.

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

## 3.3 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` | `memcpy` | `memmove` |
| `memset` | `alloca` | `bcopy` |
| `bzero` | | |

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

**#pragma intrinsic** [**(**] *func_name_list* [**)**]
**#pragma function** [**(**] *func_name_list* [**)**]
**#pragma function ( )**

The `intrinsic` pragma enables intrinsic treatment of a function. When the `intrinsic` pragma is turned on, the compiler understands how the functions work, thereby generating more efficient code. A declaration for the function must be in effect at the time the pragma is processed.

The `function` pragma disables the intrinsic treatment of a function. A function pragma 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 `intrinsic` pragma for the functions shown in Table 3-1. To enable the pragma, 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 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.

## 3.4 The #pragma linkage Directive

The C compiler (using the `–newc` flag) supports a `#pragma linkage` directive that 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 3.9).

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 ‘‘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 R0. The second parameter is a structure type with two floating-point members, which are passed in 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 (the 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);
}
```

## 3.5 The #pragma member_alignment Directive

By default, the compiler aligns structure members on natural boundaries. Use the `#pragma [no]member_alignment` {*`specifier`*} preprocessor directive (supported by the C compiler using the `–newc` flag) to determine the byte alignment of structure members.

This pragma has the following formats:

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

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 instance, an `int` variable is aligned on the next longword boundary; a `short` variable is aligned on the next word boundary.

Where the `#pragma [no]member_alignment` directives allow you to choose between natural and byte alignment, the `pragma pack` directive allows you to specify structure member alignment on byte, word, longword, or quadword boundaries. See Section 3.7 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.

The `#pragma member_alignment save` and `#pragma member_alignment restore` directives can be used to save the current state of the member_alignment (including pack alignment) and to restore 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.

## 3.6 The #pragma message Directive

The `#pragma message` directive (supported by the C compiler using the `–newc` flag) controls the issuance of individual diagnostic messages or groups of diagnostic messages. The use of this pragma overrides any command-line flags that may affect the issuance of messages.

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

**#pragma message** [ **enable | disable** ] **(***message-list***)**
**#pragma message** [ **save | restore** ]

`enable | disable` *`message-list`*

- `enable` – Enables issuance of the messages specified in the message list.
- `disable` – Disables issuance of the messages specified in the message list.
- *`message-list`*

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

  - A single message identifier.  Use the `-verbose` flag on the `cc` command to obtain the message identifier.
  - The name of a message group:
    - `ALL` – Messages in the compiler
    - `CHECK` – Messages about potentially poor coding practices
    - `PORTABLE` – Messages about portability
  - A single message identifier enclosed in parentheses.
  - A message group name enclosed in parentheses.
  - A comma-separated list of message identifiers or group names, freely mixed, enclosed in parentheses.

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

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.

`save | restore`

- `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` flags are useful primarily within header files.

## 3.7 The #pragma pack Directive

The C compiler uses the `pack` pragma to change the alignment restrictions on all members of the structure. The `pack` pragma must come prior to the entire structure definition because it acts on the whole structure. The syntax of this pragma is as follows:

**#pragma pack (*n*)**

The *n* is a number (such as 1, 2, or 4) that specifies that subsequent structure members 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* flag to the `cc` command.

## 3.8 The #pragma pointer_size Directive

This 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 } | { restore | save }**

The keywords `long` and `64` set all pointer sizes as 64-bits in all declarations that follow this directive, until the compiler encounters another

#pragma pointer_size directive.

The keywords short and 32 set all pointer sizes as 32-bits in declarations that follow this directive, until the compiler encounters another #pragma pointer_size directive.

The save keyword saves the current pointer size and the restore keyword restores the saved pointer size. 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.

The use of short pointers is restricted to DEC C++ and the C compilers resident on Digital UNIX. Programs should not attempt to pass short pointers from C++ routines to routines written in any language other than the C programming language. Also, DEC 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 (using the –newc flag) 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.

## 3.9 The #pragma use_linkage Directive

After defining a special linkage with the #pragma linkage directive, as described in Section 3.4, use the #pragma use_linkage directive (supported by the C compiler using the –newc flag) to associate the linkage with a function.

This pragma has the following format:

**#pragma use_linkage** *linkage-name* **(***routine1, routine2, ...***)**

*linkage-name*
This is the name of a linkage previously defined by the #pragma linkage directive.

*routine1*, *routine2*, *...*
These are the names of functions that you want associated with 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 R16 instead of the usual location (R0). The `parameters` option indicates that the three parameters passed to `sub` should be passed in R16, R17, and R19.

## 3.10 The #pragma weak Directive

The C compiler uses the `weak` pragma to define 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.

# Shared Libraries 4

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 discusses the following topics:

- Overview of shared libraries
- Resolving symbols
- Linking with shared libraries
- Turning off shared libraries
- Creating shared libraries
- Working with private shared libraries
- Using quickstart
- Debugging programs linked with shared libraries
- Using shared libraries from programs
- Protecting shared library files
- Shared library versioning
- Shared library restrictions

## 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.

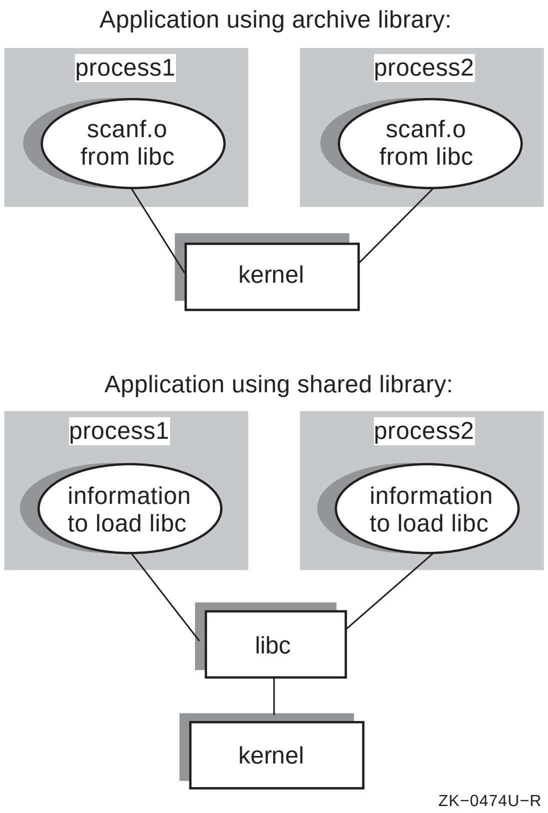
From a user perspective, the use of shared libraries is transparent. In addition, you can build your own shared libraries and make them available to other users. Most object files and archive libraries can be made into shared libraries. See Section 4.5 for more information on which files can be made 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 the `ld`(1) reference page.
- 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 end with the suffix `.a`.

Figure 4-1 illustrates 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`)
- Search path of the run-time loader (`/sbin/loader`)
- Name resolution

- Options to the `ld` command to determine behavior regarding unresolved external symbols

### 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 to the `ld` command.

### 4.2.2 Search Path of the Loader

Unless otherwise directed, the run-time loader (`/sbin/loader`) follows the same search path as the linker (`ld`). 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 it defines before searching the list of directories discussed in Section 4.2.1.

You can set the LD_LIBRARY_PATH variable by using 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 this 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 the previous section. In addition, you can use the _RLD_ROOT environment variable to alter the search path of the run-time loader. For more information, see the loader(5) reference page.

### 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 (ld) 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 were 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 to determine the behavior of the system regarding unresolved symbols, see Section 4.2.4. The following sequence is followed to resolve 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, executable program A is linked against shared library B and shared library D. Library B is linked against library C as shown in the following diagram:

```
      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:

- When building executable programs, an unresolved symbol produces an error by default. The link fails and the output file is not marked executable.
- When building shared libraries, an unresolved symbol produces only a warning message by default.

You can control the behavior of the linker by using the following flags to the `ld` command:

`–expect_unresolved` *`pattern`*

This flag specifies that any unresolved symbols matching *`pattern`* are neither displayed nor treated as warnings or errors. This flag can occur multiple times on a link command line. The patterns use shell wildcards

(?, *, [, ]) and must be quoted properly to prevent expansion by the shell. See sh(1), csh(1), and ksh(1) for more information.

–warning_unresolved
: This flag specifies that all unresolved symbols except those matching the –expect_unresolved pattern produce warning messages. This mode is the default for linking shared libraries.

–error_unresolved
: This flag causes the linker to print an error message and return a nonzero error status when a link is completed with unresolved symbols other than those matching the –expect_unresolved pattern. This mode is the default for linking executable images.

## 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 flags to the cc command to allow flexibility in determining the search path for a shared library. For example, you can use the –L*dir* flag with the cc command to change the search path by adding *dir* before the default directories, as 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 flag first with no arguments. Then, specify it again with the directory to search, followed by the –l flag 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 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 flag 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 flag. 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 flag tells the linker to resolve symbols using only shared libraries. The –lc flag 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 flag tells the linker to link all the objects from the archive library old.a. The –none flag tells the linker to turn off the –all flag. Note that the –no_archive flag applies to the resolution of the –lc flag 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, take the following 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 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.

In order 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 objects depended upon must be quickstarted.
- All objects depended upon 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 the `fixso`(1) reference page for additional 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, you should give each shared library you build a unique quickstart address 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` flag 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 these 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` flag 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 system-wide `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
```

Of course, 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. You should mention all libraries that are depended upon on the link line. After all 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 Tracking Down Quickstart Problems Manually

To determine why an executable program is not quickstarting, you can use the `fixso` utility as 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 quickstartable.

   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 unresolvable symbols. Make sure that the `ld` flags `-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 flag `–transitive_link` to the `ld` flags used when the executable program is built.

2. The executable program’s dependencies must be quickstartable.

   Get a list of an executable program’s dependencies:

   ```
   % 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 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 are not quickstartable. 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 system-wide, 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 flag 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 fixso can be used 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 flag 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. Refer to 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 were 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` flags to put the data section in the same 8-megabyte 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 avoid any unauthorized sharing, enter the following 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 binarily incompatible changes that might occur in shared libraries are the following:

- Removal of documented interfaces

  For example, if the `malloc()` function in `libc.so` were 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` were 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` were 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 binarily 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 a binarily 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 ‘‘osf.1’’. 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 version ‘‘osf.1’’ 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 binarily 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 flags used to effect version control of shared libraries.

### Table 4-1: Linker Flags that Control Shared Library Versioning

| Flag | 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.<br>If a shared library is built with this flag, 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.<br>This flag is only useful when building a shared library (with `–shared`). |
| `–exact_version` | Sets a flag 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.<br>This flag 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`*flag 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` flag, 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

Digital 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.

Digital UNIX shared libraries use a colon-separated list of version identifiers to provide the versioning features normally attained through minor versions.

The sequence of library revisions that follows illustrates 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 sequence of library revisions that follows illustrates 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 a binarily 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` flag to identify older versions of the shared libraries used by the application.

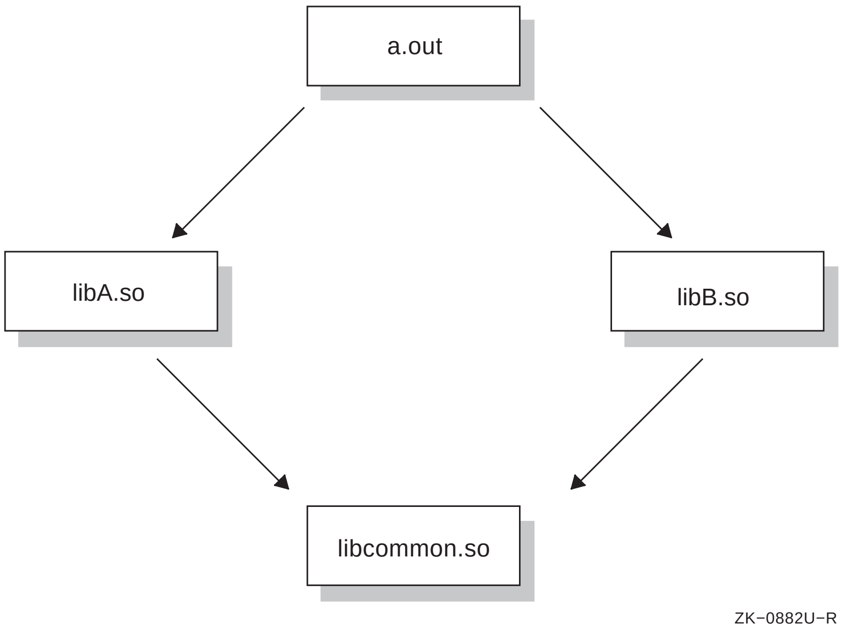
The linker issues a warning when you link an application with more than one version of the same shared library. In some cases, the multiple version dependencies of an application or shared library will not be noticed until it is loaded for execution.

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` flag 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 illustrated 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 | — | — |

Presumably `libA.so` has been linked against a previous version of `libcommon.so`. At that time the rightmost version identifier of `libcommon.so` was ‘‘v1’’. `libB.so` has been linked against the `libcommon.so` shown here. Because `libcommon.so` 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 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 ld flag –exact_match, 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

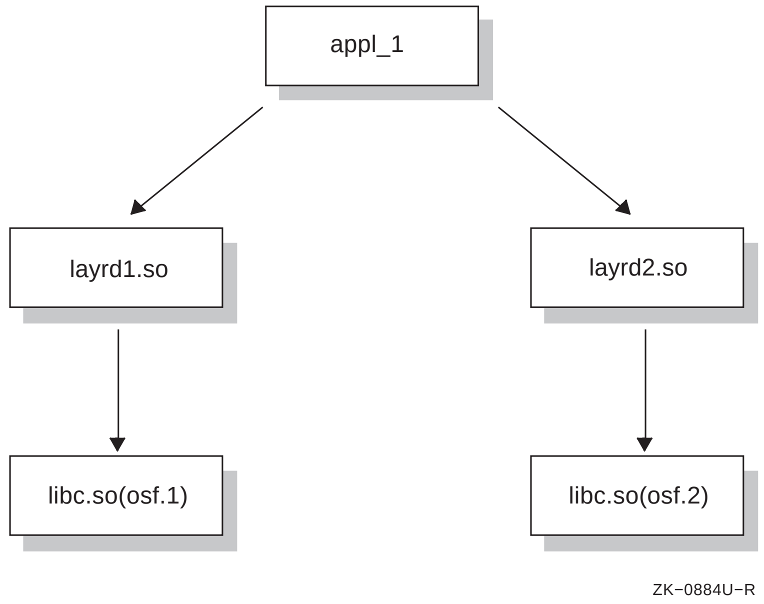
Like the linker, the loader must distinguish between valid and invalid uses of multiple versions of shared libraries:

- 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 illustrate shared object dependencies that will result in multiple dependency errors. Version identifiers are shown in parentheses.

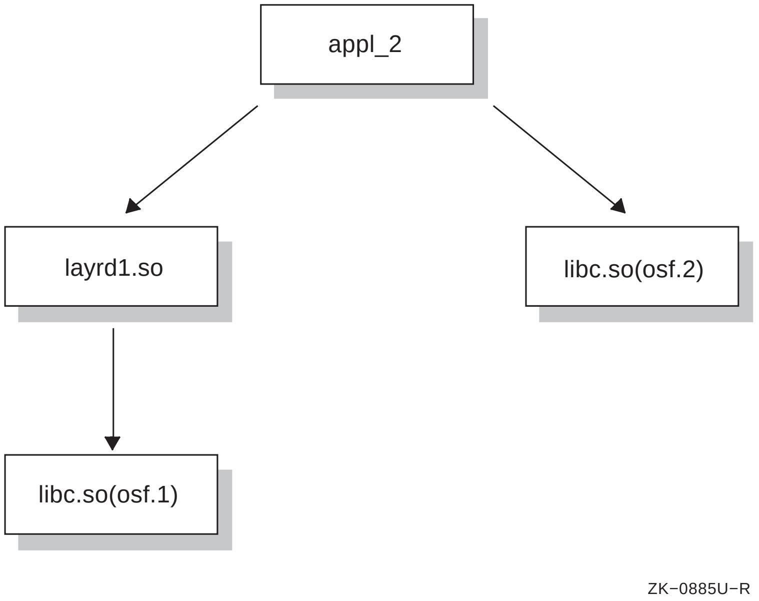
In Figure 4-3, an application uses two layered products that are built with incompatible versions of the base system.

**Figure 4-3: Invalid Multiple Version Dependencies Among Shared Objects: Example 1**



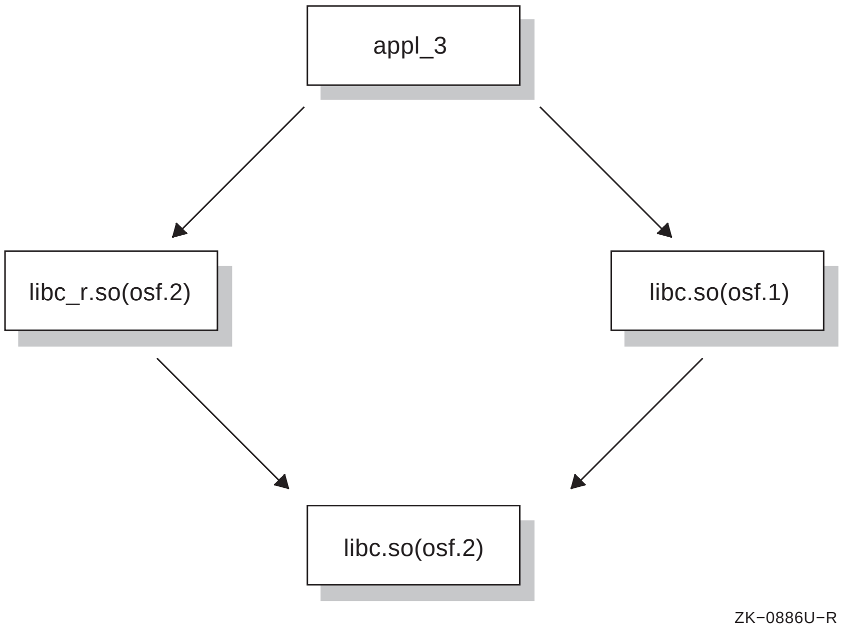
In Figure 4-4, an application is linked with a layered product that was built with an incompatible version of the base system.

**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.

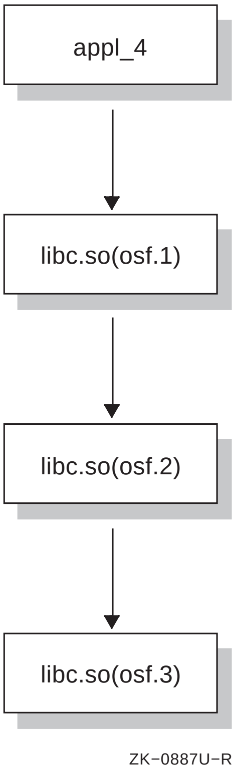
**Figure 4-5: Invalid Multiple Version Dependencies Among Shared Objects: Example 3**



The following figures show valid uses of multiple versions of shared libraries.

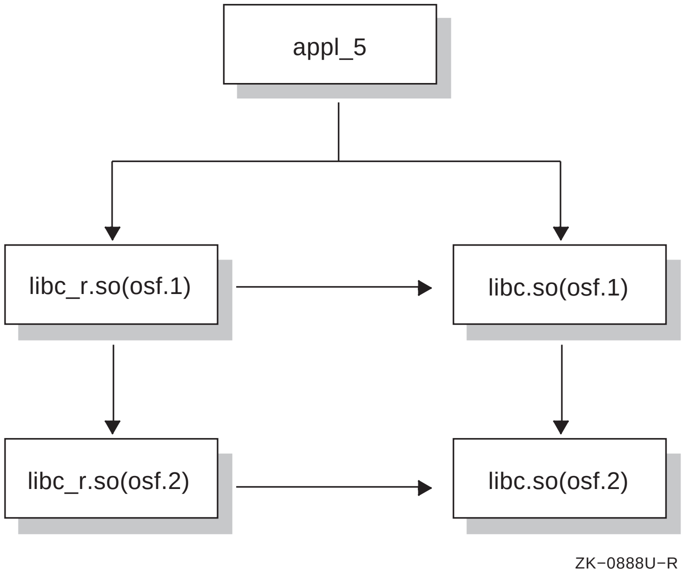
In Figure 4-6, an application uses a backward-compatibility library implemented as a partial shared library.

**Figure 4-6: Valid Uses of Multiple Versions of Shared Libraries: Example 1**



In Figure 4-7, an application uses two backward compatibile 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 Digital 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`).

# Debugging Programs with dbx 5

The dbx debugger is a tool for source level debugging. The debugger can be used with C, Fortran, Pascal, assembly language, and machine code. After invoking dbx, you can issue dbx commands that control and trace execution, display variable and expression values, and display and edit source files. The dbx debugger is a command-line program.

The ladebug debugger, an alternate debugger, provides both command-line and graphical user interfaces. In addition to supporting some languages that are not supported by dbx, the ladebug debugger also supports features for debugging multithreaded programs. For more information about ladebug, see the *Ladebug Debugger Manual*.

This chapter provides information on the following topics:

- General debugging considerations
- How to run the dbx debugger
- What you can specify in dbx commands
- How to control dbx and enter dbx commands using options provided by the dbx monitor
- How to examine source code and machine code
- How to control the execution of the program you are debugging
- How to set breakpoints
- How to examine the state of a program
- How to debug a running process
- How to debug multithreaded processes and multiple asynchronous processes

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; certain words in uppercase indicate variables for which specific rules apply. These words 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 illustrates 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 5.12.

## 5.1 General Debugging Considerations

The following sections introduce the 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 might not need to read these sections.

### 5.1.1 Why Use 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 What Are 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's 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 flags 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 flag.

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, take the following steps:

1. Set a breakpoint where you think the problem is happening — for example, 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 Avoiding Pitfalls

The debugger cannot solve all problems. For example, if your program's logic is incorrect, 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 might 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 make 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, in order 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` flag.

## 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 use the debugger, specify the –g flag at compilation time. With this flag set, the compiler inserts into the program symbol table information that the debugger uses to locate variables. With the –g flag 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 flag. 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 useful to recompile the program with –g. Note that the debugger does not warn you if an object file was compiled without the –g flag.

Complete symbol table information is available only for programs in which all modules have been compiled with the –g flag. Other programs will have symbol table information only for symbols that are either referenced by or defined in modules compiled with the –g flag.

**Note**

> Any routines in shared library applications in which breakpoints are to be set must be compiled with the –g flag. If the –g flag 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 issue 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 `dbx` and the optional parameters.

After invocation, `dbx` sets the current function to the first procedure of the program.

The `dbx` command has the following syntax:

**dbx** [ *flags* ] [ *objfile* [ *corefile* ] ]

*`flags`*
: Several of the most important flags supported by the `dbx` command line are shown in Table 5-2.

*`objfile`*
: The name of the executable file of the program that you want to debug. If *`objfile`* is not specified, `dbx` uses `a.out` by default.

*`corefile`*
: Name of a core dump file. If you specify *`corefile`*, `dbx` lists the point of program failure. The dump file holds an image of memory at the time the program failed. Use `dbx` commands to get a stack trace and look at the core file code. The debugger displays information from the core file, not from memory as it usually does.

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 Flags

| Flag | 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. Unless you specify this flag 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 4.6.1). |
| `–i` | Invokes `dbx` in interactive mode. With this flag set, `dbx` does not treat source lines beginning with number signs (#) as comments. |
| `–k` | Maps memory addresses. This flag is useful for kernel debugging. (For information on kernel debugging, see `krash`(8) and the manual *Kernel Debugging*.) |
| `-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` flag and standard input is not a terminal. If the program executes successfully, `dbx` prompts you for input. |

The following example invokes `dbx` with no flags. 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], %, `[]`, `–>` |

Table Notes:

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 `$`. The types of constants that are acceptable as input to `dbx` are shown in Table 5-6. Constants that are output from `dbx` are displayed by default as decimal values.

**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 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, you would enter the following command:

```
% setenv LINEEDIT
```

From the Bourne or Korn shells, you would 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 given position up to the cursor. If preceded by a numerical parameter whose value is greater than the current cursor position, deletes from cursor up to 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. |

**Table 5-7: (continued)**

| Command | Function |
|---|---|
| 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. |
| 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 illustrates 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 | `$asynch_interface` | 0 | Controls whether `dbx` is, or can be, configured to control multiple asynchronous processes. Incremented by 1 when a process is attached; decremented by 1 when a process terminates or is detached. Can also be set by user. If 0 or negative, asynchronous debugging is disabled. |
| B | `$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. |

### Table 5-8: (continued)

| Type | Name | Default | Description |
|---|---|---|---|
| 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 display of only real instructions when debugging in `pixie` mode. |
| 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 in strings that `dbx` prints for pointers to strings. |
| 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 zero disables it. |

### Table 5-8: (continued)

| 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 | `$printwhilestep` | 0 | For use with the `step` [*n*] and `stepi` [*n*] instructions. A nonzero value specifies that all *n* lines or instructions should be displayed. A zero value specifies that only the last line or instruction should be displayed. |
| B | `$printwide` | 0 | Specifies wide (useful for structures or arrays) or vertical format for displaying variables. A nonzero value indicates wide format; zero indicates vertical format. |
| S | `$prompt` | ‘‘`dbx)`’’ | Sets the prompt for `dbx`. |
| B | `$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. |

## Table 5-8: (continued)

| Type | Name | Default | Description |
|---|---|---|---|
| B | `$regstyle` | 1 | Specifies the type of register names to be used. A value of 1 specifies hardware names. A zero specifies software names as defined by the file `regdefs.h`. |
| B | `$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 | `$sigtramp` | ‘‘`_sigtramp`’’ | Tells `dbx` the name of the code called by the system to invoke user signal handlers. |
| B | `$stop_all_forks` | 0 | 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. |

**Table 5-8: (continued)**

| Type | Name | Default | Description |
|---|---|---|---|
| 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. |
| 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 illustrates 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. (Notice that it disappears from the alias list.)

### 5.5.4 Monitoring Debugging Session Status

Use the `status` command to check 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

To delete breakpoints and stop the recording of input and output, use the `delete` command. Deleting a breakpoint or stopping recording removes the pertinent items from the status list produced by the `status` command. To disable breakpoints without deleting them, use the `disable` command. 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 illustrates 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 the 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 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 1994
% exit
   .
   .
   .
(dbx) sh date
Tue Aug 9 17:29:34 EDT 1994
(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 flag 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 5.11 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 and 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, thus changing 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 illustrates 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 2-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.

```
(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 issuing 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 forms:

`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 ready to execute 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 ready to execute the next sequential line in the named procedure.

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 address.

`cont to` LINE
`conti to` ADDRESS

Continues until the specified source code line or machine address.

`cont in` PROCEDURE
`conti in` PROCEDURE

Continues until the specified procedure.

`cont` SIGNAL
`conti` SIGNAL

Continues from the current line or machine instruction after receiving the specified signal.

`cont` SIGNAL `to` LINE
`conti` SIGNAL `to` ADDRESS

Continues until the specified line or address after receiving the specified signal.

`cont` SIGNAL `in` PROCEDURE
`conti` SIGNAL `in` PROCEDURE

Continues until the specified procedure and sends the specified signal.

The following example shows 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 `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<br>
`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 issued against a program that is executing.

The `patch` command has the following form:

`patch` VAR = EXP<br>
`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

Although 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, 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 name 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 issued 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 issue 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 create 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

For debugging programs written in high-level languages, the stop command sets breakpoints to stop at a line, when a variable changes or a specified condition is true, or in a procedure. 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 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.

Use the delete command to remove 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 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<br>
`tracei` VAR
: Lists the specified variable after each source line or machine instruction is executed.

`trace` [VAR] `at` LINE<br>
`tracei` [VAR] `at` ADDRESS
: Lists the specified variable at the specified line or instruction.

`trace` [VAR] `in` PROCEDURE<br>
`tracei` [VAR] `in` PROCEDURE
: Lists the specified variable in the specified procedure.

`trace` [VAR] `at` LINE `if` EXP<br>
`tracei` [VAR] `at` ADDRESS `if` EXP
: Lists the variable at the specified source code line or machine 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<br>
`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]      if (argc < 2) {
(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

Use the `when` command to control 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)
```

The backslashes in this example represent line continuation. The actual output from `catch` and `ignore` is a single line.

**1** Displays the catch list.

**2** Displays the ignore list.

**3** Adds `KILL` to the catch list and removes `KILL` from the ignore list.

## 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 listing 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] |

Table Note:

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***/***countmode*

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. |

**Table 5-9: (continued)**

| Mode | Display Format |
|---|---|
| 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 reexamine 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 such purposes 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

Use the record input command to record debugger input. Use the playback input command to repeat 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 Debugging a Running Process

The `dbx` debugger can be used 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 flag `–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 but will more likely be 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 the active processes under 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: `-->`

## 5.11 Debugging Multithreaded Applications

The `dbx` debugger provides four 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.

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`.

Using the `tlist` command, you can see all the threads, with their IDs, that are currently in your program. 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 `tset` command has the following form:

`tset [EXP]`
: Choose a thread to be the current thread. The `EXP` argument is the hexadecimal ID of the desired thread.

The `tstack` command lists the stacks of all the 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:

`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 DECthreads product is installed on your system, you can gain access to the DECthreads 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 tick0, 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 = %d0,errno);
thread 0x81c62e80 stopped at   [msg_rpc_trap:75 +0x8,0x3ff808edf10]
 Source not available
(dbx) cont

Program terminated normally
```

```
(dbx) tlist
(dbx) quit
```

## 5.12 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 also 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, 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.

## 5.13 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;
```

### Example 5-1: (continued)

```
        line1.length = i;
        line1.linenumber = curlinenumber++;
        prnt(&line1);
    }
}

void prnt(pline)
LINETYPE *pline;
{
    fprintf(stdout,"%3d. (%3d) %s",
        pline->linenumber, pline->length, pline->string);
    fflush(stdout);
}
```

# Checking C Programs with lint 6

You can use the `lint` program to ensure that C programs do not contain syntax errors and to verify that the programs do not contain data type errors. This chapter describes most of the checking operations performed by `lint`, including the following:

- Program flow checking
- Data type checking
- Variable and function checking
- Migration checking
- Portability checking
- Creating a lint library
- Understanding lint error messages

See `lint`(1) for a complete list of `lint` options.

## 6.1 Overview of the lint Program

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.

The `lint` command has the following syntax:

**lint** [ *options* ] [ *file* **...** ]

*`options`*
Options to control `lint` checking operations.

The `cc` driver flags, `–std`, `–std0`, and `–std1` are available as options to `lint`. These flags affect the parsing of the source as well as the selection of the `lint` library to use. Selecting either the `–std` or `–std1` flags turns on ANSI parsing rules in `lint`.

When you use the `–MA` `lint` flag, `–std1` is used for the C preprocessing phase and `ANSI_C_SOURCES` is defined using the `–D` preprocessor flag. The following table describes the action `lint` takes for each flag:

| Lint Option | Pre-processor 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 | `llib-lc.ln` |

Table Note: EXTD is Extended C language, also known 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 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` flag 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 flags 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 option flags.

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 the following conditions:

- A loop that cannot be exited at the bottom
- A loop that cannot be entered at the top
- Infinite loops such as:

  ```
  while(1)
  for (;;)
  ```

Some programs that include these types of loops may produce correct results. These types of loops can cause problems, however.

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. Use the -O flag to the cc command when compiling the program to eliminate the resulting object code inefficiency, so that these extra statements are not important. Use the –b flag 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 *Programming Support Tools*.)

## 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 converts data types within this group automatically 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 `x`'s with pointers to `x`'s.

### 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 the –h options for the lint command line control the writing of warning messages about casts. If neither of these flags 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 flags 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 flag 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:

  /*VARARGS *n**/

  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 flags.

- 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 Using 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 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 64-bit Digital UNIX operating system. The –Q option provides support for checking programs written for ULTRIX and DEC OSF/1 Version 1.0 that you are migrating to 64-bit systems.

Because the –Q option disables checking for most other programming problems, you should 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 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. The –N option may be used on large source files to improve performance.

## 6.8 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 flag, not by using the –lport flag.

### 6.8.1 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.

### 6.8.2 Bit Field Uses

Bit fields may also 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.

### 6.8.3 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` flag.

The `-p` flag tells `lint` to change all external symbols to lowercase and limit them to six characters while checking the input files. The messages produced indicate the terms that may need to be changed.

### 6.8.4 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 illustrate 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
  ```

## 6.9 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.

### 6.9.1 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 occurs frequently 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` flag.

### 6.9.2 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` writes a message.

### 6.9.3 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` flag with the `lint` program to prevent `lint` from checking for conflicting declarations.

## 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. Perform the following steps to create a new `lint` library:

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` flag 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 flags 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 pratice and the message does not usually indicate a problem. The following program illustrates 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 flag `QWAIT` or `QNOWAIT` 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, but 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 inadvertantly 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 illustrates 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 illustrate 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 illustrates the type of code that produces 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 illustrates 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

%
```

As you can see, 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 illustrates this situation:

```
% 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 are dependent 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 only be suppressed 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 Digital 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 6.12.1), 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:<br>• Partially elided initialization [a]<br>• Static function %s not defined or used [a] |
| c | Comparisons with unsigned values. Suppresses:<br>• Comparison of unsigned with negative constant<br>• Degenerate unsigned comparison<br>• Unsigned comparison with 0? |
| d | Declaration consistency. Suppresses:<br>• External symbol type clash for %s<br>• Illegal member use: perhaps %s.%s [b]<br>• Incomplete type for %s has already been completed<br>• Redeclaration of %s<br>• Struct/union %s never defined [b]<br>• %s redefinition hides earlier one [a b] |
| h | Heuristic complaints. Suppresses:<br>• Constant argument to NOT [d]<br>• Constant in conditional context [d]<br>• Enumeration type clash, op %s<br>• Illegal member use: perhaps %s.%s [c]<br>• Null effect [f]<br>• Possible pointer alignment problem, op %s [e]<br>• Precedence confusion possible: parenthesize! [g]<br>• Struct/union %s never defined [c]<br>• %s redefinition hides earlier one [c] |
| k | K&R type code expected. Suppresses:<br>• Argument %s is unused in function %s [h]<br>• Function prototype not in scope [h]<br>• Partially elided initialization [h]<br>• Static function %s is not defined or used [h]<br>• %s may be used before set [b c]<br>• %s redefinition hides earlier one [b c]<br>• %s set but not used in function %s [h] |
| l | Assign long values to non-long variables. Suppresses:<br>• Conversion from long may lose accuracy<br>• Conversion to long may sign-extend incorrectly |
| n | Null-effect code. Suppresses:<br>• Null effect [b] |
| o | Unknown order of evaluation. Suppresses: |

**Table 6-1: (continued)**

| Warning Class | Description of Class |
|---|---|
| | • Precedence confusion possible: parenthesize! [b]<br>• %s evaluation order undefined |
| `p` | Various portability concerns. Suppresses:<br>• Ambiguous assignment for non-ansi compilers<br>• Illegal cast in a constant expression<br>• Long in case or switch statement may be truncated in non-ansi compilers<br>• Nonportable character comparison<br>• Possible pointer alignment problem, op %s [b]<br>• Precision lost in assignment to (sign-extended?) field<br>• Precision lost in field assignment<br>• Too many characters in character constant |
| `r` | Return statement consistency. Suppresses:<br>• Function %s has return(e); and return;<br>• Function %s must return a value<br>• main() returns random value to invocation environment |
| `S` | Storage capacity checks. Suppresses:<br>• Array not large enough to store terminating null<br>• Constant value (0x%x) exceeds (0x%x) |
| `u` | Proper usage of variables and functions. Suppresses:<br>• Argument %s unused in function %s [a]<br>• Static function %s not defined or used [a]<br>• %s set but not used in function %s [a]<br>• %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. |
| `C` | Constants occurring in conditionals. Suppresses:<br>• Constant argument to NOT [b]<br>• Constant in conditional context [b] |
| `D` | External declarations are never used. Suppresses:<br>• Static %s %s unused |
| `O` | Obsolescent features. Suppresses:<br>• Storage class not the first type specifier |
| `P` | Prototype checks. Suppresses:<br>• Function prototype not in scope [a]<br>• Mismatched type in function argument<br>• Mix of old and new style function declaration |

**Table 6-1: (continued)**

| Warning Class | Description of Class |
|---|---|
| | • Old style argument declaration [a]<br>• Use of old-style function definition in presence of prototype |
| R | Detection of unreachable code. Suppresses:<br>• Statement not reached |

Table notes:

a. You can also suppress this message by deactivating the `k` warning class.

b. You must also deactivate the `h` warning class to suppress this message.

c. You must also deactivate the `d` warning class to suppress this message.

d. You must also deactivate the `C` warning class to suppress this message.

e. You must also deactivate the `p` warning class to suppress this message.

f. You must also deactivate the `n` 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.

### 6.12.1 Generating Function Prototypes for Compile-Time Detection of Syntax Errors

In addition to correcting the various errors reported by the `lint` program, Digital recommends 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.

# Debugging Programs with Third Degree 7

Third Degree is an Atom tool. It performs memory access checks and memory leak detection of C and C++ programs at run time. It accomplishes this by using Atom to instrument executable objects. Instrumentation is the process of inserting instructions into existing executable objects to perform program analysis. See Chapter 9 or `atom`(1) for details on Atom.

Third Degree instruments the entire program, adding code to perform run-time checks for all of its data references. The instrumented program locates many occurrences of the worst types of bugs in C and C++ programs: array overflows, memory smashing, and errors in the use of the `malloc` and `free` functions. It also helps you determine the allocation habits of your application by listing the heap and finding memory leaks.

Except for being larger and running slower than the original application and having its uninitialized data filled with a special pattern, the instrumented program runs like the original. The Atom instrumentation code logs all specified errors and generates the requested reports.

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 use Third Degree to instrument programs using other memory allocators, such as the `mmap` function, but it will not check accesses to the memory thus obtained.

  Third Degree detects and forbids calls to the `brk` function. Furthermore, if your program allocates memory by partitioning large blocks it obtained by using the `sbrk` function, Third Degree may not be able to precisely identify memory blocks in which errors occur.

- Applications using POSIX threads (`pthread`) interfaces and applications using a supported coroutine package. Most coroutine packages are supported. If your application uses a custom threads or coroutine package, you may not be able to use Third Degree. See Section 7.1.2 for details.

## 7.1 Running Third Degree on an Application

You invoke the Third Degree tool by using the `atom` command, as follows:

```
% atom app -tool third
```

In this example, `app` is the name of an application. When it is run, the instrumented version of the application (`app.third`) behaves exactly like the original application (`app`), with the following exceptions:

- The code is larger and runs more slowly because of the additional instrumentation code that is inserted.
- 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 `object_padding` option in the `.third` file. (See Section 7.2.1 for a description of the `.third` customization file.)
- To detect errant use of uninitialized data, Third Degree initializes all otherwise uninitialized data to a special pattern. 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 disable Third Degree's initialization with the `–uninit_heap` and `–uninit_stack` option in the `.third` customization file.

The instrumented version of the application generates a log file (`app.3log`) containing information about allocated objects and potential leaks.

**Note**

> Third Degree writes `.3log` messages in a format similar to that used by the C compiler. If you use `emacs` or a similar editor that automatically points, in sequence, to each compilation error, you can use the same editor to follow Third Degree errors. In `emacs`, compile with a command such as `cat app.3log`, and step through the Third Degree errors as if they were compilation errors.

You can control the name used for the output log file by specifying one of the following flags to the `–toolargs` flag on the `atom` command line that invokes the Third Degree tool:

`–pids`
: Appends the process identification number to the log file name.

`–nopids`
: Does not append the process identification number to the log file name. This is the default.

`–dirname`*`fname`*

Specifies the directory path in which Third Degree creates its log file.

Depending upon the flag supplied to Third Degree in the `atom` command's `–toolargs` flag, the log file's name will be as follows:

| Flag | Filename | Use |
|---|---|---|
| `–nopids` | `app.3log` | Default |
| `–pids` | `app.12345.3log` | Include pid |
| `–dirname /tmp` | `/tmp/app.3log` | Set directory |
| `–dirname /tmp –pids` | `/tmp/app.12345.3log` | Set directory and pid |

## 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 the `–non_shared` or `–call_shared` flags.

The `atom` command provides the following flags to allow you to determine which shared libraries are instrumented by Third Degree:

`–all`

Instruments all statically loaded shared libraries in the shared executable.

`–excobj` *`objname`*

Excludes the named shared library from instrumentation. You can use the `–excobj` flag more than once to specify several shared libraries.

`–incobj` *`objname`*

Instruments the named shared library. You can use the `–incobj` flag more than once to specify several shared libraries.

When Atom finishes instrumenting the application, the current directory contains an instrumented version of each specified shared library. The instrumented application uses these versions of the libraries. Define the `LD_LIBRARY_PATH` environment variable to tell the instrumented application where the instrumented shared libraries reside.

By default, Third Degree does not instrument any of the shared libraries used by the application; 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 terminates stack traces at the first uninstrumented procedure. 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 with all of its shared libraries and run the new instrumented version.

### 7.1.2 Using Third Degree with Threaded Applications

Third Degree supports applications that use threads. To instrument a threaded application, add the `–env threads` flag to the `atom` command line that invokes the Third Degree tool.

## 7.2 Step-by-Step Example

Assume that you must debug the small application represented by the following source code (`ex.c`):

```
/* ex.c */
#include <assert.h>;

int Bug() {
    int q;
    return q;           /* q is uninitialized */
}

long* Booboo(int n) {
    long* t = (long*) malloc(n * sizeof(long));
    t[0] = Bug();
    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 = Booboo(20);
    t = Booboo(4);
    free(t);            /* already freed */
    exit(0);
}
```

### 7.2.1 Customizing Third Degree

An optional customization file named `.third` is used to turn on and off various capabilities of the Third Degree tool and to set the tool's internal parameters. Third Degree looks for a `.third` file first in the local directory, then in your home directory. The `.third` customization file is further discussed throughout this chapter and its syntax is described in the `third`(5) reference page.

If you do not specify a `.third` customization file, Third Degree uses its default settings:

- List memory errors
- Detect leaks at program exit
- No heap history

## 7.2.2 Modifying the Makefile

Add the following entry to the application’s Makefile:

```
ex.third: ex
        atom ex -tool third -o ex.third
```

Build `ex.third` as follows:

```
> make ex.third
atom ex -tool third -o ex.third
> ex.third
```

Now run the instrumented application `ex.third` and check the log `ex.3log`.

## 7.2.3 Examining the Third Degree Log File

The `ex.3log` file contains several sections, described in the following sections.

### 7.2.3.1 Copy of the .third File

If you supplied a `.third` customization file, Third Degree copies it to the log file. The short customization file used in this example requests a summary of the contents of heap-allocated memory blocks when the program finishes:

```
//////////////  begin .3rd  ///////////////////
-----------------------------------------------
heap_history    yes
-----------------------------------------------
//////////////  end  .3rd ///////////////////
```

### 7.2.3.2 List of Runtime Memory Access Errors

The types of errors that Third Degree can detect at runtime 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 process id 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 may well 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 Bug 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 Bug
    Bug                            ex.c, line 6
    Booboo                         ex.c, line 11
    main                           ex.c, line 20
    __start                        crt0.s, line 370
```

- The following log entry indicates that an error occurred 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.

```
---------------------------------------------------- ruh -- 1 --
ex.c: 12: reading uninitialized heap at byte 8 of 160-byte block
    Booboo                         ex.c, line 12
    main                           ex.c, line 20
    __start                        crt0.s, line 370

This block at address 0x38000000f10 was allocated at:
    malloc                         malloc.c, line 585
    Booboo                         ex.c, line 10
    main                           ex.c, line 20
    __start                        crt0.s, line 370
```

- 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
    Booboo                         ex.c, line 14
    main                           ex.c, line 20
    __start                        crt0.s, line 370

This block at address 0x38000000f10 was allocated at:
    malloc                         malloc.c, line 585
    Booboo                         ex.c, line 10
    main                           ex.c, line 20
    __start                        crt0.s, line 370
```

- 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 `Booboo` (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                           malloc.c, line 833
    main                           ex.c, line 22
    __start                        crt0.s, line 370

This block at address 0x380000011a0 was allocated at:
    malloc                         malloc.c, line 585
    Booboo                         ex.c, line 10
    main                           ex.c, line 21
    __start                        crt0.s, line 370

This block was freed at:
    free                           malloc.c, line 833
    Booboo                         ex.c, line 15
    main                           ex.c, line 21
    __start                        crt0.s, line 370
```

### 7.2.3.3 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.

```
---------------------------------------------------------------
---------------------------------------------------------------
Searching for new leaks in heap after program exit

160 bytes in 1 object were found:

160 bytes in 1 leak (including 1 super leak) created at:
    malloc                        malloc.c, line 585
    Booboo                        ex.c, line 10
    main                          ex.c, line 20
    __start                       crt0.s, line 370
```

Upon examining the source, it is clear that the first call of `Booboo` did not free the memory object, nor was it freed anywhere else in the program. Moreover, no pointer to this object exists anywhere in the program, so it qualifies as a super leak. 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 object in the structure is a leak, but losing the pointer to the root is the real cause of the leak. Because all objects but the root still have pointers to them, albeit only from other leaks, only the root will be identified as a super leak, and therefore the likely cause of the memory loss.

#### 7.2.3.4 Heap History

When heap history is enabled, Third Degree collects information about dynamically allocated memory. It collects this information for every object that is freed by the application and for every object 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 (5% written) in 1 objects created at:
    malloc                         malloc.c, line 585
    Booboo                         ex.c, line 10
    main                           ex.c, line 20
    __start                        crt0.s, line 370

Contents:
     0: ..ii....................................

----------------------------------------------------------------
32 bytes allocated (25% written) in 1 objects created at:
    malloc                         malloc.c, line 585
    Booboo                         ex.c, line 10
    main                           ex.c, line 21
    __start                        crt0.s, line 370

Contents:
     0: ..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 one long (8 bytes) in each array was set to a valid value, resulting in the written ratios of 8/160 = 5% and 8/32=25% shown. The character map, with one character for each 32-bit word in the object, shows that the initialized value was the second long in each of the arrays.

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.

### 7.2.3.5 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. The first two entries give the final (maximum) sizes of the heap and stack at the end of the program. The last two entries give the text and static data areas for the program and any shared libraries.

```
-----------------------------------------------------------------
-----------------------------------------------------------------
  memory layout at program exit
              heap      81920 bytes [0x38000000000-0x38000014000]
             stack       2224 bytes [0x11ffff750-0x120000000]
           ex data      23168 bytes [0x140000000-0x140005a80]
           ex text     262144 bytes [0x120000000-0x120040000]
```

## 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.

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.

Memory errors include the following (as represented by a three-letter abbreviation):

| **Name** | **Error** |
|---|---|
| `ror` | Reading out of range: neither in heap, stack, or static area |
| `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 providing a `.third` customization file containing the `ignore` option. 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`(5) for more details.

### 7.3.1 Fixing Errors and Retrying an Application

If Third Degree reports many write errors from your instrumented program, you should fix the first few errors and 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 initializes all blocks to a nonzero value.

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 supplying a `.third` customization file containing the `uninit_heap no` and `uninit_stack no` options. Using the `uninit_stack no` option disables the initialization of newly allocated stack memory that Third Degree normally performs on each procedure entry. Similarly, the `uninit_heap no` option disables the initialization 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.

**Notes**

> Do not use the `uninit_heap no` and `uninit_stack no` options under normal operation. They hamper Third Degree's ability to detect a program's use of uninitialized data.

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 supplying a `.third` customization file including the `signals no` 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.

In order 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, supply a `.third` customization file including a `use` option. This allows Third Degree to find the source files contained in other directories. Specifying the `use` option with no arguments clears the search path. 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 pointer exists. The object can no longer be accessed and can no longer be used or freed. It is useless and will never go away.

Third Degree finds memory leaks by using a simple trace-and-sweep algorithm. Starting from a set of roots (the currently active stack and static area), 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, create a `.third` configuration file that specifies the `pointer_mask` option. The `pointer_mask` option lets you specify a mask that is applied as an AND operator against every potential pointer. For example, if you use the top 3 bits of pointers as flags, specify a mask of 0x1fffffffffffffff. See `third`(5) for additional information on `.third` configuration files.
- 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.

### 7.4.2 Reading Heap and Leak Reports

You can supply `.third` configuration file 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`(5) for details).

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 one-megabyte 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 a `.third` configuration file containing the `object_stack_depth` or `leak_stack_depth` option. (See `third`(5) for further description of the `.third` configuration file.) These options set 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 `.third` configuration file `object_min_percent` or `leak_min_percent` option. (See `third`(5) for further description of the `.third` configuration file.) These options define 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. When an object is allocated, listed, or shrunk through a call to the `realloc` function, it 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 a program, providing a `.third` configuration file specifying the `all leaks before exit every 1` option line will result in Third Degree not finding 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) `.third` configuration file, 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 memory allocations. The number of allocations depends on the characteristics of the application being instrumented. Use a `.third` configuration file specifying the following options:

```
no leaks at_exit
new leaks before proc_name every 10000
```

See `third`(5) for further description of the `.third` configuration file.

### 7.4.4 Interpreting the Heap History

When you instrument this program, providing a `.third` configuration file specifying the `heap_history yes` option line allows Third Degree to generate a heap history for the program. A heap history allows you to see how the program used dynamic memory during its execution. You can use this feature, for instance, 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 at any point up to this point of the program’s execution
- Total number of bytes allocated at this point of the program’s execution
- Total number of objects that have been allocated up to this point of the program’s execution
- 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 allocated at this point 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 `history_size` option in the `third` configuration file.

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, or the heap; or was zero in every object. |
| 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 100MB, it does not mean that at any point in time 100MB of heap storage were used by the allocated objects. It is a cumulative figure; it indicates that this point has allocated 100MB over the lifetime of the program. This 100MB 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%. 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. If they are kept on a free list, however, they will only be found 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
    __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 address of the unnamed procedure.

More frequently, the file name or line number is unavailable because the program's symbol table is incomplete. 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.

If the lack of symbolic information is hampering your debugging, consider recompiling the program with more symbolic information. For C and C++ programs, recompile with the `–g` flag and link without the `–x` flag.

## 7.6 Validating Third Degree Error Reports

The following spurious errors may occur in rare instances:

- Modifications to bit fields in optimized code are occasionally reported as uses of uninitialized data. This situation usually occurs in initializations of arrays of items smaller than 32 bits or in initializations of packed structures, as in the following 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 */
  }
  ```

- Third Degree initializes 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.

- Storing the special uninitialized value into memory and subsequently reading it (though the value is neither a valid pointer, a floating-point number, a remarkable integer, nor ASCII characters).

If you think that you have found a false positive, you can verify it by using the disassembler (`dis`) 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.

## 7.7 Undetected Errors

Third Degree can fail to detect real errors, such as the following:

- Errors in logical operations on quantities smaller than 32 bits can go undetected, for example:

  ```
  short Small() {
      short x;
      x &= 1;
      return x;
  }
  ```

  This programming practice may be considered an error if the program depends on the least significant bit of `x`. It may not be considered an error if the program depends only on the most significant bits.

- 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, supply a `third` customization file that includes the `object_padding` option.

- Third Degree may not be able to detect if the application walks past the end of an array by fewer than 8 bytes. Because Third Degree brackets objects in the heap by "guard words," it will miss small array bounds errors. 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.

- 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.

# Profiling Programs to Improve Performance 8

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.

This chapter discusses the following topics:

- Using the `prof` program
- Using the `gprof` program
- Using the `pixie` and `hiprof` Atom tools
- Using the `uprofile` and `kprofile` tools
- Selecting profiling information to display
- Using feedback files
- Using profiling environment variables
- Using `monitor` routines
- Profiling multithreaded applications

## 8.1 Profiling Methods

Profiling methods include:

- Program counter (PC) sampling, a technique that periodically interrupts your program and logs the value of the PC. The `prof` and `gprof` tools use PC sampling to produce a statistical sample showing which portions of code consume the most time. The `gprof` tool also produces call graphs, which show the relationship of calling and called routines.
- Basic block counting, a technique that inserts profiling code at key points in your program. It produces a count of the number of times each instruction executes.

To select an appropriate profiling method for an application, you must take into consideration the following factors:

- The statistics that you want to collect and examine (for example, CPU usage, call counts, call cost, memory usage, and I/O operations)

- The level at which you need to collect these statistics (for example, at a procedure level or at an instruction level).
- Whether you must profile the shared libraries used by the application as well as its executable.
- The method that you use to collect the profiling data. Certain collection methods require that you compile and/or link the application's sources in a special way. Others allow you to run a utility that inserts instrumentation code into an existing program. Still others retrieve information from the CPU's performance counters while the uninstrumented program is running.
- The tool that you use to display the profiling data. Depending on the information that you need, you can choose from three tools that display previously collected profiling information. Each tool supports multiple data collection methods.

The profiling data display tools, and their respective data collection methods, include the following:

`prof`

Prints a profile of statistics per procedure.

The `prof` tool supports the following data collection methods:

- Compiling or linking with the `–p` flag

  The `–p` flag supports the profiling of shared libraries, but requires you to at least relink the program. It collects only CPU statistics using PC sampling

- Using the `uprofile` tool

  The `uprofile` tool profiles user code. It does not support the profiling of shared libraries. It does not require you to relink the program and collects either CPU statistics or other information.

- Using the `kprofile` tool

  The `kprofile` tool profiles the running operating system kernel. It does not require you to relink the program and collects either CPU statistics or other information.

`prof –pixie`

Prints a profile showing the number of times each procedure, source line, or instruction is executed. The `prof –pixie` tool supports the following basic block counting profiling data collection method:

- Using the `pixie` Atom tool (that is, the `atom –tool pixie` command) to instrument the program's basic blocks.

  The `pixie` Atom tool supports the profiling of shared libraries and does not require you to relink the program. It supports the `prof` tool's instruction-level profiling and true cycle-count estimation.

`gprof`

Produces call-graph profile data showing the effects of calling routines on called routines as well as other information.

The `gprof` tool supports the following data collection methods:

- Compiling with the `–pg` flag

  The `–pg` flag does not allow the profiling of shared libraries. It requires that you recompile the program's sources and uses an apportioned call cost method to determine a given procedure's cost to its callers.

- Using the `hiprof` Atom tool (that is, the `atom –tool hiprof` command) to instrument the program

  The `hiprof` Atom tool supports the profiling of shared libraries and does not require you to recompile or relink. To determine a given procedure's cost to its callers, it supports both the apportioned call cost method and the measured call cost method.

You can also use the `monitor` routines to perform PC-sampling on a specified address range in a program. For more information on using `monitor` routines, see Section 8.13 and `monitor`(3).

## 8.2 Profiling Tools Overview

Table 8-1 provides a concise overview of the profiling tools available in the Digital UNIX operating system.

**Table 8-1: Profiling Tools**

| Tool | Use |
|---|---|
| PC-sampling/ `prof` | Link application with `–p;` analyze results with `prof`; see `prof`(1) and `monitor`(3). |
| Call-arcs/ `gprof` | Compile and link with `–pg;` analyze results with `gprof`; see `gprof`(1) and `monitor`(3). |
| `pixstats` | Additional postprocessor for pixified program output; see `pixstats`(1). |
| `uprofile`/ `kprofile` | Run application under `uprofile` or `kprofile`; requires `pfm` driver to be installed; analyze results with `prof`; see `uprofile`(1), `kprofile`(1), and `pfm`(7). |

**Table 8-1: (continued)**

| Tool | Use |
|---|---|
| Atom toolkit | Programmable debug/performance analysis tool. Example tools are contained in `/usr/lib/cmplrs/atom/examples`; see `atom`(1) and other Atom reference pages for programming interface. |
| `pixie` | Atom-based basic block profiler; analyze results with `prof`; see `pixie`(5). |
| `hiprof` | Atom-based call-arc analyzer; analyze results with `gprof`; see `hiprof`(5). |
| `third` | Atom-based memory error/leak detection tool, Third Degree; generates text output. See `third`(5). |

All profiling tools work on call-shared and nonshared applications.

### 8.2.1 PC-Sampling

Statistical PC-sampling for the program is useful for diagnosing high CPU-usage procedures in the program and it supports both threads and shared libraries.

Interface summary:

```
% cc -p *.o -o program      # Link with libprof1.a
% program                   # Run program to collect data
% prof program              # Process the mon.out file
```

### 8.2.2 gprof

The `gprof` tool provides procedure call information coupled with statistical PC-sampling. This is useful for determining which routines are called most frequently and from where. The `gprof` tool also gives a flat profile for CPU-usage on the routines. It supports threads and call-shared programs, but does not support shared libraries.

Using the `gprof` tool, you can retrieve information from `libc.a` and `libm.a` because these two libraries are compiled with the `–pg` flag. Other Digital-supplied libraries are not compiled with `–pg`, so calling information on these other system libraries is not available.

Interface summary:

```
% cc -pg *.c -o program    # Compile and link with -pg
% program                  # Run program to collect data
% gprof program            # Process the gmon.out file
```

### 8.2.3 uprofile and kprofile

The `uprofile` and `kprofile` tools use the performance counters on the Alpha chip. They do not collect information on shared libraries. By default, both tools collect cycles for the program. The performance data produced by these tools is processed with the `prof` command. See `uprofile`(1) and `kprofile`(1) for more information.

### 8.2.4 Atom Toolkit

The Atom toolkit consists of a programmable instrumentation tool and several packaged tools. Examples are included in the `/usr/lib/cmplrs/atom/examples` directory that demonstrate how to develop instrumentation and analysis code. The instrumentation part of the tool instructs Atom on where to insert calls to analysis routines in the program. When the program is run, the analysis routines are entered and data collection is performed as prescribed by the Atom tool specified on the `atom` command.

Atom does not work on programs built with the `–om` flag.

Interface summary:

```
% atom -tool toolname program
% program.tool
```

Postprocessing is tool-dependent. See Chapter 9 for details on Atom.

### 8.2.5 pixie Atom tool

The Atom-based pixie is a basic block profiler that supports shared libraries and threaded applications.

Interface summary:

```
% atom -tool pixie [-env threads] program
% program.pixie[.threads]
% prof -pixie program
```

### 8.2.6 hiprof Atom tool

The `hiprof` Atom tool collects call-arc information on a program. By default, it operates like the `gprof` support provided by the `–pg` flag, but has flag-selectable options that are more powerful. The `hiprof` Atom tool supports shared libraries and threaded applications.

Interface summary:

```
% atom -tool hiprof [-env threads] program
% program.hiprof[.threads]
% gprof program program.hiout
```

### 8.2.7 Third Degree

Third Degree is a memory-leak and memory-overwrite detection tool, also based on Atom. Third Degree generates text output to a file called `program.3log`. The log contains the diagnostics that Third Degree detected (for example, reads of uninitialized heap or stack, memory overwrites, and memory leaks).

Interface summary:

```
% atom -tool third [-env threads] program
% program.third[.threads]
% cat program.3log
```

## 8.3 Profiling Sample Program

The examples in the remainder of this chapter refer to the sample program, `profsample.c`, shown in Example 8-1.

**Example 8-1: Profiling Sample Program**

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

#define LEN     100

void mult_by_scalar(double ary[], int len, double num);
void add_vector(double arya[], double aryb[], int len);
double value;
void printit(double value);

main()
{
    double ary1[LEN];
    double ary2[LEN];
    int i;
```

**Example 8-1: (continued)**

```
    for (i=0;  i<LEN;  i++) {
        ary1[i] = 0.0;
        ary2[i] = sqrt((double)i);
    }
    mult_by_scalar(ary1, LEN, 3.14159);
    mult_by_scalar(ary2, LEN, 2.71828);
    for (i=0;  i<20;  i++)
        add_vector(ary1, ary2, LEN);
}

void mult_by_scalar(double ary[], int len, double num)
{
    int i;

    for (i=0;  i<len;  i++)
     {
        ary[i] *= num;
        value = ary[i];
        printit(value);
      }
}

void add_vector(double arya[], double aryb[], int len)
{
    int i;

    for (i=0;  i<len;  i++)
      {
        arya[i] += aryb[i];
        value = arya[i];
        printit(value);
      }
}

void printit(double value)
{
        printf("Value = %f\n", value);
}
```

## 8.4 Using prof to Produce Program Counter Sampling Data

To use `prof` to obtain PC sampling data on a program, follow these steps:

1. Compile and link (or just link) using the –p option, as follows:

   ```
   % cc -c profsample.c
   % cc -p -o profsample profsample.o -lm
   ```

   You must specify the –p profiling option during the link step to obtain PC sampling information. If you have an existing application, you will not need to recompile to profile the executable program; simply relink the program using the –p option with the cc command.

   If you are building an application for the first time, you can compile and link in the same step. In the preceding example, the –lm option ensures that libm.{a,so} is used to resolve symbols that refer to math library functions.

   You might also consider compiling with one of the optimization flags to help improve the efficiency of your code, compiling with a debug flag to provide more symbolic information for the profile report, or compiling with both types of flags.

   If you are profiling a multithreaded application, use the –threads flag with the cc command. For more information on profiling multithreaded applications, see Section 8.14.

2. Execute the profiled program:

   ```
   % profsample
   ```

   You can run the program several times, altering the input data (if any) to create multiple profile data files.

   During execution, profiling data is saved in a profile data file. The default name for the profile data file is mon.out, unless you have set the environment variable PROFDIR. For more information on using PROFDIR, see Section 8.12.1

3. Run the profile formatting program prof, which extracts information from one or more profile data files and produces a tabular report:

   ```
   % prof profsample mon.out
   ```

Example 8-2 shows output produced by the prof command on the profsample.c program.

### Example 8-2: Profiler Listing for PC Sampling

```
Profile listing generated Thu May 26 13:36:14 1994 with:
   prof profsample mon.out

--------------------------------------------------------------
*  -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 14% of 0.0068 seconds

%time   seconds  cum %   cum sec  procedure (file)

 42.9    0.0029   42.9      0.00 printit (profsample.c)
 42.9    0.0029   85.7      0.01 add_vector (profsample.c)  1
 14.3    0.0010  100.0      0.01 mult_by_scalar (profsample.c)
```

**1** This sample line of output presents the following information:

- 42.9 percent of execution time was spent in `add_vector`.
- 85.7 percent of total execution time was spent cumulatively in the `printit` and `add_vector` routines.
- The name of the source file for `mult_by_scalar` is `profsample.c`

Because the `prof` program works by periodic sampling of the program counter, you might see different output when you profile the same program multiple times. A different profiling run than the preceding example of the sample program produced the following output:

```
Profile listing generated Thu May 26 13:34:00 1994 with:
   prof -procedures profsample mon.out

--------------------------------------------------------------
*  -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 17% of 0.0059 seconds

%time     seconds  cum %   cum sec  procedure (file)

 66.7      0.0039   66.7      0.00 add_vector (profsample.c)
 33.3      0.0020  100.0      0.01 printit (profsample.c)
```

## 8.5 Using gprof to Display Call Graph Information

To determine the manner in which routines call, or are called by, other routines, use the `gprof` profiling tool.

The `gprof` tool postprocesses both `hiprof` output and `–pg` output.

To use this tool, follow these steps:

1. Use the `hiprof` Atom tool to produce an instrumented version of the program:

   ```
   % atom -tool hiprof profsample
   ```

2. Execute the instrumented version of `profsample`:

   ```
   % profsample.hiprof
   ```

3. Examine the profiling data as follows:

   ```
   % gprof profsample profsample.hiout
   ```

During execution, profiling data is saved in the data file `profsample.hiout`, unless you have set the `–dirname` flag in the `HIPROF_ARGS` environment variable or on the command line.

Alternatively, you can use the following procedure to collect profiling data for the `gprof` tool:

1. Compile and link using the `–pg` option, as follows:

   ```
   % cc -pg -c profsample.c
   % cc -pg -o profsample profsample.o -lm
   ```

   You must specify the `–pg` flag with the `cc` command during both the compile and link steps to obtain call graph information.

2. Execute the program:

   ```
   % profsample
   ```

   When this method is used, profiling data is saved during execution in the data file `gmon.out`, unless you have set the `PROFDIR` environment variable. For more information on using this variable, see Section 8.12.1.

3. Run the formatting program `gprof`, which extracts information from the data file:

   ```
   % gprof profsample gmon.out
   ```

The output produced by the gprof utility comprises three sections:

- Call graph profile
- Timing profile, similar to the profile produced by prof
- Index

You can control gprof profiling by file by using the –no_pg flag to the cc command. When you use this flag, you disable gprof profiling for all objects that follow the flag on the command line. You cannot use the –no_pg flag with the –r and –shared flags to the ld command.

Example 8-3 shows output for gprof profiling of the sample program. The –b flag was used with gprof to suppress printing of the description of each output field. The descriptions are valuable, but they are lengthy and were left out due to space considerations. To see these descriptions, follow the steps to produce gprof output and write the output to a file or pipe the output through the more utility.

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. Note that for the purpose of this example output, the three sections have been separated by rows of asterisks that do not appear in the output produced by gprof. Each row of asterisks includes the name of the section. For more information on gprof flags, see the gprof(1) reference page.

### Example 8-3: Sample gprof Output

```
*********************** call graph profile *******************

granularity: each sample hit covers 4 byte(s) for 10.00%
of 0.01 seconds

                                called/total       parents
index %time  self descendents   called+self   name          index
                                called/total       children

                                                    <spontaneous>
[1]   100.0  0.00        0.01                 main [1]
             0.00        0.00    20/20          add_vector [2]
             0.00        0.00     2/2           mult_by_scalar [4]

------------------------------------------------

             0.00        0.00     20/20         main [1]      1
[2]    75.5  0.00        0.00     20          add_vector [2]  2
             0.00        0.00   2000/2200       printit [3]   3

------------------------------------------------
```

**Example 8-3: (continued)**

```
             0.00        0.00     200/2200      mult_by_scalar [4]
             0.00        0.00   2000/2200       add_vector [2]
[3]    50.0  0.00        0.00   2200          printit [3]

-----------------------------------------------

             0.00        0.00      2/2           main [1]
[4]     4.5  0.00        0.00      2           mult_by_scalar [4]
             0.00        0.00    200/2200        printit [3]

-----------------------------------------------

*********************** timing profile section ***************

granularity: each sample hit covers 4 byte(s) for 10.00%
of 0.01 seconds

  %   cumulative   self        self     total
time  seconds  seconds  calls  ms/call  ms/call  name
50.0    0.00    0.00   2200     0.00     0.00   printit [3]
30.0    0.01    0.00     20     0.15     0.37   add_vector [2]
20.0    0.01    0.00                            main [1]
 0.0    0.01    0.00      2     0.00     0.22   mult_by_scalar[4]

*********************** index section ***********************
Index by function name

   [2] add_vector            [4] mult_by_scalar
   [1] main                  [3] printit
```

**1** This line describes the relationship of the `main` routine to the `add_vector` routine. Because `main` is listed above the `add_vector` routine in the final column of this section, `main` is identified as the parent of `add_vector`. The fraction 20/20 indicates that of the 20 times that `add_vector` (the denominator of the fraction) was called, it was called 20 times by `main` (the numerator of this fraction).

**2** This line describes the `add_vector` 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 75.5% in the second column reports the total amount of time in the sample that is accounted for by the `add_vector` routine and its descendent, in this case the `printit` routine. The 20 in the `called` column indicates the total number of times that the `add_vector` routine is called.

**3** This line describes the relationship of the `printit` routine to the `add_vector` routine. Because the `printit` routine is below the `add_vector` routine in this section, `printit` is identified as the child of add_vector. The fraction 2000/2200 indicates that of the total of 2200 calls to printit, 2000 of these calls came from add_vector.

## 8.6 Using pixie for Basic Block Counting

A basic block is a set of instructions with one entry and one exit. The pixie Atom tool provides execution counts for the basic blocks of a program. With prof, the execution counts can be viewed at the instruction level.

To obtain data for basic block counting, follow these steps:

1. Compile and link. For example:

   ```
   % cc -c profsample.c
   % cc -o profsample profsample.o -lm
   ```

2. Run the pixie Atom tool. You do not have to specify a name for the output because pixie produces an output file by default with the same name as the original C source file, but with pixie appended after a period. For example, the following command causes pixie to create two files, profsample.pixie and profsample.Addrs:

   ```
   % atom -tool pixie profsample
   ```

   The profsample.pixie file is equivalent to profsample but contains additional code that counts the execution of each basic block. To create an output file with a name other than *pname*.pixie, use the –o flag followed by the name you assign to the output file.

   The profsample.Addrs file contains the address of each of the basic blocks. For more information, see pixie(5).

3. Execute the profsample.pixie file:

   ```
   % profsample.pixie
   ```

   This command generates the file profsample.Counts, which contains the basic block counts. Each time you execute the profsample.pixie file, you create a new profsample.Counts file.

4. Run the profile formatting program prof, with the –pixie flag over the profsample executable file:

   ```
   % prof -pixie profsample
   ```

   This command extracts information from profsample.Addrs and profsample.Counts and displays information in an easily readable format. Note that you do not need to specify the .Addrs and .Counts file suffixes because pixie searches by default for files containing them.

You can also run the pixstats program on the executable file profsample to generate a detailed report on opcode frequencies, interlocks, a miniprofile, and more. For more information, see pixstats(1).

**Note**

> The pixie profiling tool provided in the current version of the Digital UNIX operating system is the pixie Atom tool. If you use the syntax provided in earlier versions of the operating system to invoke pixie, a script transforms the call into a call to the pixie Atom tool. The previous version of the pixie tool can be found at /usr/opt/obsolete/usr/bin/pixie.

## 8.7 Selecting Profiling Information to Display

Depending on the size of the application and the type of profiling you request, prof may generate a very large amount of output. However, you are often only interested in profiling data about a particular portion of your application.

### 8.7.1 Limiting Profiling Display to Specific Procedures

The prof program provides the following flags to display information selectively by procedure:

```
–only
–exclude
–Only
–Exclude
–totals
```

The –only option tells prof to print only profiling information for a particular procedure. You can specify the –only option multiple times on the command line. For example, the following command displays profiling information for procedures mult_by_scalar and add_vector from the sample program:

```
% prof -only mult_by_scalar -only add_vector profsample
```

The –exclude option tells prof to print profiling information for all procedures except the specified procedure. You can use multiple –exclude flags on the command line.

The following command displays profiling information for all procedures except `add_vector`:

```
%  prof -exclude add_vector profsample
```

Do not use the `–only` and `–exclude` flags on the same command line.

Many of the `prof` utility's profiling flags print output as percentages, for example, the percentage of total execution time attributed to a particular procedure.

By default, the `–only` and `–exclude` flags 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` flags. These flags work the same as `–only` and `–exclude`, but cause `prof` to calculate percentages based only on those procedures that appear in the listing. For example, the following command omits the `add_vector` procedure from both the listing and from percentage calculations:

```
% prof -Exclude add_vector profsample
```

The `–totals` flag, used with the `–procedures` and `–invocations` listings, prints cumulative statistics for the entire object file instead of for each procedure in the object.

### 8.7.2 Including Shared Libraries in the Profiling Information

The `–all`, `–incobj`, and `–excobj` flags allows you to display profiling information for shared libraries used by the program:

- The `–all` flag causes the profiles for all shared libraries (if any) described in the data file(s) to be displayed, in addition to the profile for the executable.
- The `–incobj` flag causes the profile for the named shared library to be printed, in addition to the profile for the executable.
- The `–excobj` flag causes the profile for the named executable or shared library not to be printed.

### 8.7.3 Using pixie to Display Profiling Information for Each Source Line

The `–heavy` and `–lines` flags cause `prof` to display the total number of machine cycles executed by each source line in your application. Both of these flags require you to use basic block counting (the `–pixie` option); they do not work in PC-sampling mode.

The `–heavy` option prints an entry for every source line that was executed by your application. Each entry shows the total number of machine cycles executed by that line. Entries are sorted from the line with the most machine cycles to the line with the least machine cycles. Because this option often prints a huge number of entries, you might want to use one of the `–quit`, `–only`, or `–exclude` flags to reduce output to a manageable size.

Example 8-4 shows output generated by the following command:

```
% prof -pixie -heavy -only add_vector -only mult_by_scalar \
  -only main profsample
```

For example, you can see in Example 8-4 that line 47 of `profsample.c` in the procedure `add_vector( )` accounts for over 12 percent of the application’s total execution time. The listing also shows the size in bytes of each source line.

### Example 8-4: Prof Output by Source Line with -heavy Flag

```
Profile listing generated Fri May 27 14:09:10 1994 with:
  prof -pixie -heavy -only add_vector -only mult_by_scalar
  -only main profsample

-------------------------------------------------------------------
*  -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 (profsample.c)       48    44      22000  23.26  23.26
add_vector (profsample.c)       46    40      20000  21.15  44.41
add_vector (profsample.c)       47    24      12000  12.69  57.10
mult_by_scalar (profsample.c)   36    44       2200   2.33  59.43
main (profsample.c)             20    60       1500   1.59  61.02
mult_by_scalar (profsample.c)   34    28       1400   1.48  62.50
mult_by_scalar (profsample.c)   35    24       1200   1.27  63.77
main (profsample.c)             19    12        300   0.32  64.08
main (profsample.c)             25    48        240   0.25  64.34
add_vector (profsample.c)       41    28        140   0.15  64.48
add_vector (profsample.c)       44    12         60   0.06  64.55
add_vector (profsample.c)       50    12         60   0.06  64.61
mult_by_scalar (profsample.c)   29    28         14   0.01  64.63
main (profsample.c)             23    32          8   0.01  64.63
main (profsample.c)             22    32          8   0.01  64.64
mult_by_scalar (profsample.c)   38    12          6   0.01  64.65
mult_by_scalar (profsample.c)   32    12          6   0.01  64.66
main (profsample.c)             26    16          4   0.00  64.66
```

### Example 8-4: (continued)

```
main (profsample.c)             13    16          4   0.00  64.66
main (profsample.c)             18     8          2   0.00  64.67
main (profsample.c)             24     8          2   0.00  64.67
```

The `–lines` option is similar to `–heavy`, but it 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 from those procedures that execute the most machine cycles to those that execute the least.

Example 8-5 shows the same information as Example 8-4, but in a different format as generated by the following command:

```
% prof -pixie -lines -only add_vector -only mult_by_scalar \
 -only main profsample
```

### Example 8-5: Prof Output by Source Line with -lines Flag

```
Profile listing generated Fri May 27 14:07:28 1994 with:
   prof -pixie -lines -only add_vector -only mult_by_scalar
   -only main profsample

-------------------------------------------------------------------
*  -l[ines] using basic-block counts;                              *
*  grouped by procedure, sorted by cycles executed per procedure;*
*  '?' means that line number information is not available.      *
-------------------------------------------------------------------

procedure (file)              line bytes     cycles      %  cum %

add_vector (profsample.c)       41    28        140   0.15   0.15
                                44    12         60   0.06   0.21
                                46    40      20000  21.15  21.36
                                47    24      12000  12.69  34.05
                                48    44      22000  23.26  57.32
                                50    12         60   0.06  57.38
mult_by_scalar (profsample.c)   29    28         14   0.01  57.39
                                32    12          6   0.01  57.40
                                34    28       1400   1.48  58.88
                                35    24       1200   1.27  60.15
                                36    44       2200   2.33  62.48
                                38    12          6   0.01  62.48
main (profsample.c)             13    16          4   0.00  62.49
                                18     8          2   0.00  62.49
                                19    12        300   0.32  62.81
                                20    60       1500   1.59  64.39
                                22    32          8   0.01  64.40
                                23    32          8   0.01  64.41
```

**Example 8-5: (continued)**

```
                                   24    8          2   0.00  64.41
                                   25   48        240   0.25  64.66
                                   26   16          4   0.00  64.67
```

### 8.7.4 Limiting Profiling Display by Line

The `–quit` option reduces the amount of profiling output displayed. The `–quit` option affects the output from the `–procedures`, `–heavy`, and `–lines` profiling modes.

The `–quit` option provides three versions:

- `–quit` *n*

  The *n* refers to an integer. All lines after the *n* line are truncated.

- `–quit` *n*%

  The *n* is an integer followed by a percent sign (%). All lines after the line containing *n*% calls in the `%calls` column of the display are truncated.

- `–quit` *n*`cum%`

  The *n*`cum%` refers to an integer *n* followed by the characters `cum` (for cumulative) and a percent sign (%). All lines after the line containing *n*`cum%` calls in the `cum%` column of the display are truncated.

If you specify several modes on the same command line, the `–quit` option affects the output from each mode. For example, the `–quit` option in the following command reduces the output from both the `–procedures` and `–heavy` modes:

```
% prof -pixie -procedures -heavy -quit 20 profsample
```

This command prints only the 20 most time-consuming procedures and the 20 most time-consuming source lines. The `–quit` *n* option has no affect on the `–lines` profiling mode.

The `–quit` *n*% option restricts the output to those entries that account for at least *n*% of the total. Depending on the profiling mode, the total can refer to the total amount of time, the total number of machine cycles, or the total number of invocation counts. For example, the following command prints only those source lines that account for at least 2 percent of the application's total number of machine cycles:

```
% prof -pixie -lines -quit 2% profsample
```

The `–quit` *n*`cum%` option truncates the output after *n*% of the total has been accounted for. The definition of total depends on the profiling mode, as described in the preceding paragraph. For example, the following command prints the most heavily used source line and stops after 30 percent of the application's total number of machine cycles have been accounted for:

```
% prof -pixie -heavy -quit 30cum% sample
```

## 8.8 Using pixie to Average prof Results

A single run of a program may not produce the desired results. You can repeatedly run the version of the program created by `pixie`, varying the input with each run, and then use the resulting `.Counts` files to produce a consolidated report. For example:

1. Compile and link. Do not use the `–p` option when linking to produce an executable file for `pixie`:

   ```
   % cc -c profsample.c
   % cc -o profsample profsample.o -lm
   ```

2. Run the profiling utility `pixie`, as follows:

   ```
   % atom -tool pixie -toolargs=-pids profsample
   ```

   This command produces the `profsample.Addrs` file to be used in step 4, as well as the modified program `profsample.pixie`.

3. Delete any existing `.Counts` files, set the `PIXIE_ARGS` environment variable to `"-pids"`, and run the executable program produced by `pixie`. For example:

   ```
   % profsample.pixie
   ```

   The `–pids` option specified with the `atom -tool pixie` command in step 2 appends the process ID of the process running the executable program to the name of the `profsample.Counts` file, for example, `profsample.Counts.1753`.

4. Run the profiled program as many times as desired. Each time the program is run, a `profsample.Counts.`*<pid>* file is created.

5. Run `prof` to create the report as follows:

   ```
   % prof -pixie profsample profsample.Addrs profsample.Counts.*
   ```

   If you had run `profsample.pixie` three times, the `prof` utility would have averaged the basic block data in the three files generated by the executable (`profsample.Counts.`*<pid1>*, `profsample.Counts.`*<pid2>*, and `profsample.Counts.`*<pid3>*) to produce the profile report.

## 8.9 Analyzing Test Coverage

When you are writing a test suite for an application, you might want to know how effectively your suite tests the application. The `prof` utility provides two flags that can help you determine this. The `–zero` option prints the names of procedures that were never executed by your application. The `–testcoverage` option lists all of the source lines that were never executed by your application. Both of these flags require basic block counting.

Typically, you would perform the following steps to make use of these flags.

1. Run the `pixie` Atom tool on your application.
2. Run the results of step 1 through your test suite saving any `.Counts` files.
3. Profile your application with the `–zero` or `–testcoverage` flags and specify all of the `.Counts` files produced when you ran the test suite.

## 8.10 Merging Data Files

If the application 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. If you are using PC sampling, each run of your application produces a new `mon.out` file, or a *`program.pid`* file if you have set the `PROFDIR` environment variable. If you are using basic block counting, each run produces a new `.Counts` file.

You have two ways of displaying profiling information that is based on an average of all of these output files.

The first way is to specify the names of each profiling data file explicitly on the command line. For example, the following command prints profiling information from two profile data files:

```
% prof -procedures profsample 1510.profsample 1522.profsample
```

Keeping track of many different profiling data files, however, can be difficult. Therefore, `prof` provides the `–merge` option to combine several data files into a single merged file. When `prof` operates in `–pixie` mode, the `–merge` flag 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 profsample 1773.profsample \
    1777.profsample
```

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 profsample 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:

```
% prof -pixie -merge total.Counts a.out a.out.Addrs \
  a.out.Counts.1866 a.out.Counts.1868
```

## 8.11 Using Feedback Files

Feedback files are useful in identifying portions of a large executable program in which significant percentages of the execution occur. Without feedback, the compiler must make assumptions about call frequency based on nesting levels. These assumptions are almost never as good as actual data from a sample run. The following sections describes how to use feedback files by using the `cc` command and the `atom -tool pixie` and `prof` commands.

### 8.11.1 Generating and Using Feedback Information

Follow these steps to generate feedback information that can be used to optimize subsequent compilations:

1. Compile the source code:

   ```
   % cc -O2 -o profsample profsample.c -lm
   ```

2. Run the `pixie` Atom tool on the executable file:

   ```
   % atom -tool pixie -toolargs=-o profsample.pixie profsample
   ```

   This step creates an output executable file named `profsample.pixie` and a `prof` input file named `profsample.Addrs`.

3. Execute the program you just created:

   ```
   % profsample.pixie
   ```

   This step creates a file named `profsample.Counts`, which contains execution statistics.

4. Use prof to create a feedback file from the execution statistics:

   ```
   % prof -pixie -feedback profsample.feedback profsample
   ```

5. You can use a feedback file as input to a compilation at –O2 or –O3 optimization levels when you use the –feedback option with the cc command, as shown in the following example:

   ```
   % cc -O3 -feedback profsample.feedback -o \
     profsample profsample.c -lm
   ```

   The feedback file provides the compiler with actual execution information that can be used to improve certain optimizations, such as inlining function calls. Use a feedback file generated from a –O2 compilation for any subsequent compilations with –O2 or –O3 flags.

### 8.11.2 Using a Feedback File for Input to cord

You can also use a feedback file as input to the cord utility. The cord utility orders the procedures in an executable program to improve execution time. The following example shows how to use the –cord option as part of a compilation command with a feedback file as input:

```
% cc -O2 -cord -feedback profsample.feedback \
  -o profsample profsample.c -lm
```

Use a feedback file generated with the same optimization level as the level you use in subsequent compilations.

You can also use cord with the runcord utility. For more information, see runcord(1).

## 8.12 Using Environment Variables to Control PC-Sample Profiling

By default, the –p and –pg flags to the cc command provide the following:

- A single profile covering the whole text segment and all threads. To profile specific portions of the program, use the monitor utilities, as described in Section 8.13 and monitor(3).
- A single data file called mon.out (for –p) or gmon.out (for –pg) placed in the current directory.

The –p flag supports the profiling of shared libraries. The –pg flag and uprofile tool support the profiling of only the part of a program that is in the executable. When using these tools to generate profiling information for library routines, link your object file with the –non_shared flag to the cc command.

You can use one of the following environment variables to control profiling behavior:

- PROFDIR
- PROFFLAGS

By using these variables, you can disable aspects of default profiling behavior, including:

- Changing the name and path of profiling data files
- Controlling when profiling begins
- Controlling profiling of multithreaded applications

You can use the PROFFLAGS and PROFDIR environment variables together.

Note that these environment variables have no effect on the prof and gprof post-processors; they affect the profiling behavior of a program during its execution. These environment variables have no effect when you use the pixie Atom tool.

### 8.12.1 PROFDIR Environment Variable

By default, profiling data is collected in a data file named [g]mon.out. When you do multiple profiling runs, each run overwrites the existing [g]mon.out file. Use the PROFDIR environment variable when you want to collect PC sampling data in files with unique names. Set this environment variable as follows:

- C Shell:

  ```
  setenv PROFDIR path
  ```

- Bourne Shell:

  ```
  PROFDIR = path ; export PROFDIR
  ```

The results are saved in the file *path/pid.progname*, which resolves as follows:

*path*
: The directory path, specified with PROFDIR, identifying an existing directory.

*pid*
: The process ID of the executing program.

*progname*
: The program name.

When you set PROFDIR to a null string, no profiling occurs.

### 8.12.2 PROFFLAGS Environment Variable

By default, the profiling library libprof1.a (or libprof1_r.a, for multithreaded programs) allocates one buffer per process to record your profiling data, as well as placing the data output file in your current directory.

To disable this default behavior, set the PROFFLAGS environment variable as follows:

- C Shell:

  ```
  setenv PROFFLAGS "-disable_default"
  ```

- Bourne Shell:

  ```
  PROFFLAGS = "-disable_default"; export PROFFLAGS
  ```

When you have set PROFFLAGS to –disable_default, the default profiling support is disabled, allowing you to use the monitor calls to profile specific sections of your program for both nonthreaded and multithreaded programs. See monitor(3) and Section 8.13 for more information on using the monitor, monstartup, and moncontrol routines.

For multithreaded programs, you can allocate one buffer per thread by setting the PROFFLAGS environment variable as follows:

- C Shell:

  ```
  setenv PROFFLAGS "-threads"
  ```

- Bourne Shell:

  ```
  PROFFLAGS = "-threads"; export PROFFLAGS
  ```

When you have set PROFFLAGS to –threads, a separate file is produced for each thread and is named *pid.sid.progname*, which is resolved as follows:

*pid*
The process identification of the program.

*sid*
The sequence number of the thread, which depends on the order in which the threads were created.

*progname*
The name of the program being profiled.

You can use the –threads and –disable_default flags together to control profiling of your program when you use the monitor routines.

You can also set the PROFFLAGS environment variable to include or exclude profiling information:

`setenv PROFFLAGS "-all"`
: Causes the profiles for all shared libraries (if any) described in the data file(s) to be displayed, in addition to the profile for the executable.

`setenv PROFFLAGS "-incobj` *lib_name*`"`
: Causes the profile for the named shared library to be printed, in addition to the profile for the executable.

`setenv PROFFLAGS "-excobj` *lib_name*`"`
: Causes the profile for the named executable or shared library not to be printed.

## 8.13 Using monitor Routines to Control Profiling

The default profiling behavior on Digital 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:

`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. You can set the `PROFFLAGS` environment variable to `–disable_default` to prevent profiling from beginning when your program executes. Then, you can use the monitor routines to begin profiling after the first call to monitor or monstartup.

You can disable the default naming of the profiling data file by using the PROFDIR environment variable. For more information on using this environment variable, see Section 8.12.1.

Example 8-6 demonstrates how to use the monstartup and monitor routines within a program to begin and end profiling.

### Example 8-6: 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.
      */

     monstartup(__start,&_etext);

        for(i=0;i<10;i++)
             domath( );

     /* Stop profiling and write the profiling output file. */

     monitor(0);

}
 domath( )
{
   int i;
```

**Example 8-6: (continued)**

```
   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 8-7.

**Example 8-7: 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 */
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.
       */
```

### Example 8-7: (continued)

```
         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 8-8 illustrates how to use the `monitor_signal` routine.

### Example 8-8: Using monitor_signal() to Profile Non-Terminating 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;
```

**Example 8-8: (continued)**

```
    /*
     * 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);
}
```

## 8.14 Profiling Multithreaded Applications

Profiling multithreaded applications is essentially the same as profiling non-threaded applications. However, to profile multithreaded applications, you must compile your program with the `–pthread` or `–threads` flag to the `cc` command. Specifying one of these flags and either the `–p` or `–pg` flag enables the thread profiling library, `libprof1_r.a`.

The default case for profiling multithreaded applications is to provide one sampling buffer for all threads. In this case, you get sampling across the entire process and you get one output file comprising sampling data from all threads. Depending on whether you use the `–p` or `–pg` flag, your output file will be named `mon.out` or `gmon.out`, respectively.

To get a separate buffer and a separate output file for each thread in your program, use the environment variable `PROFFLAGS`. Set `PROFFLAGS` to `–threads`, as shown in the following example:

```
setenv PROFFLAGS "-threads"
```

The profiling data file will be named according to the following convention:

```
pid.sid.progname
```

In the preceding example, *`pid`* is the process id of the program, *`sid`* corresponds to the order in which the thread was created, *`progname`* is your program name.

If the application controls profiling by using the `monitor` routines, `sid` corresponds to the order in which profiling was started for the thread.

If you use the `monitor( )` or `monstartup( )` calls in a threaded program, you must first set `PROFFLAGS` to `"-disable_default -threads"`, giving 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` flag, 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.

# Using and Developing Atom Tools 9

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 instruction and 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 the program are to be instrumented. Atom is independent of any compiler and language as it operates on object modules that make up the complete program.

Atom, as provided in the Digital UNIX operating system, provides the following:

- A set of prepackaged tools that may be used to instrument applications for profiling or debugging purposes. Use the following command to apply one of these tools to a given application:

  **atom** *application_program* **–tool** *toolname* **–env** *environment*

- A command interface and a collection of instrumentation routines that may be used to create custom Atom tools. Use the following command to create a custom-designed Atom tool:

  **atom** *application_program* *instrumentation_file* *analysis_file*

The `atom`(1) reference page describes both forms of the `atom` command.

This chapter contains the following sections:

- Section 9.1 describes the prepackaged Atom tools and how to use them.
- Section 9.2 discusses how you can develop specialized Atom tools.

## 9.1 Using Prepackaged Atom Tools

The Digital UNIX operating system provides and supports the Atom tools listed in Table 9-1.

### Table 9-1: Supported Prepackaged Atom Tools

| Tool | Description |
|---|---|
| Third Degree (`third`) | Performs memory access checks and detects memory leaks in an application. The Third Degree Atom tool is described in Chapter 7 and in the `third`(5) reference page. |
| `hiprof` | Produces a flat profile of an application that shows the execution time spent in a given procedure and a hierarchical profile that shows the execution time spent in a given procedure and all its descendants. The `hiprof` Atom tool is described in Chapter 8 and `hiprof`(5). |
| `pixie` | Partitions an application into basic blocks and counts the number of times each basic block is executed. The `pixie` Atom tool is described in Chapter 8 and `pixie`(5). |

The Digital UNIX operating system provides the unsupported Atom tools listed in Table 9-2 as examples for programmers developing custom-designed Atom tools. These tools are distributed in source form to illustrate Atom's programming interfaces. Some of the tools are further described in Section 9.2.

### Table 9-2: Example Prepackaged Atom Tools

| Tool | Description |
|---|---|
| `branch` | Instruments all conditional branches to determine how many are predicted correctly. |
| `cache` | Determines cache miss rate if application runs in 8K direct-mapped cache. |
| `dtb` | Determines the number of dtb (data translation buffer) misses if the application uses 8KB pages and a fully associative translation buffer. |
| `dyninst` | Provides fundamental dynamic counts of instructions, loads, stores, blocks, and procedures. |

**Table 9-2: (continued)**

| Tool | Description |
|---|---|
| `inline` | Identifies potential candidates for inlining. |
| `iprof` | Prints the number of times each procedure is called as well as the number of instructions executed (dynamic count) 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 (dynamic count) by each procedure. |
| `ptrace` | Prints the name of each procedure as it is called. |
| `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. |

## 9.2 Developing Atom Tools

An Atom tool consists of the following:

- An instrumentation file – Modifies the application to which it is applied by adding calls at well-defined locations to tool-specific analysis procedures.
- An analysis file – Defines the procedures and data structures required to implement the tool's functionality.

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 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, 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 procedure call to an analysis routine. The tool can specify that the procedure call be made before or after an object, procedure, basic block, or instruction.

### 9.2.1 The ATOM Command Line

The command line used to apply Atom tools to an application is described completely in the `atom`(1) reference page. This section describes the command line and its most commonly used arguments and flags.

The `atom` command line has two forms:

**atom** *application_program* **–tool** *toolname* [ **–env** *environment* ] [ *flags...* ]

> This form of the `atom` command is used to build an instrumented version of an application program using a prepackaged Atom tool.
>
> This form requires the `–tool` flag and accepts the `–env` flag. It does not allow either the *`instrumentation_file`* or the *`analysis_file`* parameter.
>
> The `–tool` flag identifies the prepackaged Atom tool to be used. By default, Atom searches for prepackaged 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` flag identifies any special environment (for instance, *`threads`*) in which the tool is to operate. The set of environments supported by a given tool is defined by the tool's creator and listed in the tool's documentation. Atom displays an error if you specify an environment that is undefined for the tool. The prepackaged tools allow you to omit the `–env` flag to obtain a general-purpose environment.

**atom application_program** *instrumentation_file* [ *analysis_file* ] [ flags... ]

> This form of the `atom` command is used to apply a tool that instruments an application program. This form requires the *`instrumentation_file`* parameter and accepts the *`analysis_file`* parameter.
>
> The *`instrumentation_file`* parameter specifies the name of a C source file or an object module that contains the Atom tool's instrumentation procedures. By convention, most instrumentation files have the suffix `.inst.c` or `.inst.o`.

The *analysis_file* parameter specifies the name of a C source file or an object module that contains the Atom tool's analysis procedures. Note that you do not need to specify an analysis file if the instrumentation file does not call analysis procedures. By convention, most analysis files have the suffix `.anal.c` or `.anal.o`.

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.tool
```

### 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 issuing 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.tool
```

With the exception of the `–tool` and `–env` flags, both forms of the `atom` command accept any of the remaining flags described in the `atom`(1) reference page. The following are some flags that deserve special mentioning:

`–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 flag increases the performance of the instrumented application by a factor of 2 (at the expense of some increase in application size). The default behavior is for Atom not to apply these optimizations.

`–debug`

Allows you to debug instrumentation routines by causing Atom to transfer 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"
```

–g

Causes Atom to build the analysis procedures with debugging symbol table information, allowing you to run instrumented applications under a symbolic debugger. Atom assumes that the application itself runs correctly, allowing debugger commands to be used only on analysis procedures. For example:

```
% atom hello ptrace.inst.c ptrace.anal.c -o hello.ptrace -g
```

```
% 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] }
```

–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
```

### 9.2.2 Atom Instrumentation Routine

Atom invokes a tool's instrumentation routine on a given application program when that program is specified as the *application_program* parameter to the atom command, and either of the following is true:

- The tool is a prepackaged tool specified as an argument to the –tool flag of an atom command. By default, Atom looks for prepackaged tools in the /usr/lib/cmplrs/atom/tools and /usr/lib/cmplrs/atom/examples directories.
- The file containing the instrumentation routine is specified as the *instrumentation_file* parameter of an atom command.

The instrumentation routine contains the code that traverses the objects, procedures, basic blocks, and instructions to locate instrumentation points; adds calls to analysis procedures; and builds the instrumented version of an application.

As described in the atom_instrumentation_routines(5) reference page, 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 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, thus allowing 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` flag 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, 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 procedure 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.

#### 9.2.3.1 Navigating Within a Program

The Atom application navigation routines, described in the `atom_application_navigation`(5) reference page, allow an Atom tool's instrumentation routine to find locations in an application at which to add calls to analysis procedures.

- 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 `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. Similarly, the GetBlockProc routine returns a pointer to the procedure that contains the specified basic block, and 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 the atom_object_management(5) reference page, 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 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 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 the atom_application_query(5) reference page, allow an instrumentation routine to obtain static information about a program and its objects, procedures, basic blocks, and instructions.

The GetAnalName routine 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. For example, multiple cache simulators might share a single instrumentation file but each have a different analysis file.

The GetProgInfo routine returns the number of objects in a program.

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, basic blocks, or instructions it contains; its object ID; 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 filename 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 instrumention 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* 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 such information about the procedure as the number of basic blocks or instructions it contains, its procedure ID, its lowest or highest source line number, or an indication if its address has been taken. |
| `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 lists the routines that provide information about a procedure’s basic blocks:

### Table 9-5: 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. The block ID is unique to this basic block within its containing object. |
| `IsBranchTarget` | Indicates if the block is the target of a branch instruction. |

Table 9-6 lists the routines that provide information about a basic block’s instructions:

### Table 9-6: Atom Instruction Query Routines

| Routine | Description |
|---|---|
| `GetInstBinary` | Returns a 32-bit binary representation of the assembly language instruction. |
| `GetInstClass` | Returns the instruction class (for instance, floating-point load or integer store) as defined by the *Alpha Architecture Reference Manual*. An Atom tool uses this information to determine instruction scheduling and dual issue rules. |
| `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. |
| `InstPC` | Returns the compile-time program counter (PC) of the instruction. |
| `InstLineNo` | Returns the instruction’s source line number. |

**Table 9-6: (continued)**

| Routine | Description |
|---|---|
| `IsInstType` | Indicates whether the instruction is of the specified type (load instruction, store instruction, conditional branch, or unconditional branch). |

### 9.2.3.4 Resolving Procedure Names and Call Targets

Resolving procedure names and subroutine targets is trivial for nonshared programs because all procedures are contained in the same object. However, the target of a subroutine branch in a call-shared program could be in any object.

The Atom application procedure name and call target resolution routines, described in the `atom_application_resolvers`(5) reference page, allow an Atom tool's instrumentation routine to find a procedure by name and to find a target procedure for a call site:

- The `ResolveTargetProc` routine attempts to resolve the target of a procedure call.
- The `ResolveNamedProc` routine returns the procedure identified by the specified name string.
- The `ReResolveProc` routine completes a procedure resolution if the procedure initially resided in an unbuilt object.

### 9.2.3.5 Adding Calls to Analysis Routines to a Program

The Atom application instrumentation routines, described in the `atom_application_instrumentation`(5) reference page, add arbitrary procedure calls at various points in the application:

- 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`, `AddCallBlock`, and `AddCallInst`. Atom provides facilities for passing integers and floating-point numbers, arrays, branch condition values, effective addresses, cycle counters, as well as procedure arguments and return values.
- 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 `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.

### 9.2.4 Atom Description File

An Atom tool's description file, as described in the `atom_description_file`(5) reference page, identifies and describes the tool's instrumentation and analysis files. It can also specify the flags 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 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 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 flags 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 any system call or library function, 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.

#### 9.2.5.1 Input/Output

An analysis procedure that uses the standard I/O library should take care to explicitly close file descriptors before the instrumented application exits. The standard I/O library buffers read and write requests to optimize disk accesses. It flushes an output buffer to disk either when it is full or when a procedure calls the fflush function. If the instrumented application exits before an analysis procedure properly closes its output file descriptors, the procedure's output may not be completely written.

Some Atom tool analysis procedures may print results to stdout or stderr. Because the file descriptors for these I/O streams are closed when an instrumented application calls the exit function, an analysis routine that is called from an instrumentation point set by a call to the ProgramAfter routine can no longer send output to either. Analysis procedures written in C++ must also take care when using the cout and cerr functions. Because these streams are buffered by the class library, an analysis routine must call cout.flush() or cerr.flush() before the instrumented application exits.

#### 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, this is exactly the behavior than an Atom tool expects. 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 9-2), 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, 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 for only the child process.

### 9.2.6 Determining the Instrumented PC from an Analysis Routine

The Atom Xlate routines, described in Xlate(5), allow you to determine the instrumented PC for selected instructions. You can use these functions to build a table that translates an instruction's PC in the instrumented application to its PC in the uninstrumented application.

To enable analysis code to determine the instrumented PC of an instruction at runtime, an Atom tool's instrumentation routine must select the instruction and place it into an address translation buffer (XLATE).

An Atom tool's instrumentation routine creates and fills the address translation buffer by calling the CreateXlate and AddXlateAddress routines, respectively. An address translation buffer can only hold instructions from a single object.

The AddXlateAddress routine adds the specified instruction to an existing address translation buffer.

An Atom tool's instrumentation 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 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 runtime addresses in the object corresponding to the specified address translation buffer and the compile-time addresses.
- The XlateAddr routine returns the instrumented runtime address for the instruction in the specified position of the specified address translation buffer. Note that the runtime 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, issue the following instruction:

```
% 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 succede 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);
}
```

## 9.2.7 Sample Tools

This section describes the basic tool building interface by using three simple examples: procedure tracing, instruction profiling, and data cache simulation.

### 9.2.7.1 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`.

```
#include <stdio.h>
#include <cmplrs/atom.inst.h>  1

unsigned InstrumentAll(int argc, char **argv)  2
{
   Obj *o; Proc *p;
   AddCallProto("ProTrace(char *)");  3
   for (o = GetFirstObj(); o != NULL; o = GetNextObj(o)) {  4
     if (BuildObj(o) return 1;  5
     for (p = GetFirstObjProc(o); p != NULL; p = GetNextProc(p)) {  6
       const char *name = ProcName(p);  7
       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 procedure names, such as those defined as `static`, may not be available. (Compiling applications with the `–g1` flag 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.

Once the instrumentation and analysis files are specified, the tool is complete. To illustrate the application of this tool, we compile and link the following application:

```
#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
% 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.

#### 9.2.7.2 Profile Tool

The prof 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, prof creates a file called prof.out that contains a profile of the number of instructions that are executed in each procedure.

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 prof tool's instrumentation procedure (prof.inst.c) performs these tasks:

```
#include <stdio.h>
#include <cmplrs/atom.inst.h>

unsigned InstrumentAll(int argc, char **argv)
{
  Obj *o; Proc *p; Block *b; Inst *i;
  int n = 0;
  AddCallProto("OpenFile(int)");  1
  AddCallProto("Count(int,int)");
  AddCallProto("Print(int,char *)");
  AddCallProto("CloseFile()");
  for (o = GetFirstObj(); o != NULL; o = GetNextObj(o)) {  2
    if (BuildObj(o)) return (1);  3
    for (p = GetFirstObjProc(o); p != NULL; p = GetNextProc(p)) {  4
      const char *name = ProcName(p);  5
      if (name == NULL) name = "UNKNOWN";
17       for (b = GetFirstBlock(p); b != NULL; b = GetNextBlock(b)) {  6
18         AddCallBlock(b,BlockBefore,"Count",n,  7
             GetBlockInfo(b,BlockNumberInsts));
19       }
20       AddCallProgram(ProgramAfter,"Print",n,name);  8
21       n++;  9
22     }
23     WriteObj(o);  10
24   }
25   AddCallProgram(ProgramBefore,"OpenFile",n);  11
26   AddCallProgram(ProgramAfter,"CloseFile");  12
27   return (0);
28 }
```

**1** Defines the interface to the analysis procedures.

**2** Loops through each object in the program.

**3** Builds an object.

**4** Loops through each procedure in the object.

**5** Determines the procedure name.

**6** Loops through each basic block in the procedure.

**7** Adds a call to the `Count` analysis procedure before any of the instructions in this basic block are executed. The argument types of the `Count` are defined in the prototype on Line 9. 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 `Count` analysis procedure adds the number of instructions in the basic block to a per-procedure data structure.

**8** Adds a call to the `Print` analysis procedure to the end of the program. The `Print` analysis procedure prints a line summarizing this procedure’s instruction use.

**9** Increments the procedure index.

**10** Writes the object file.

**11** 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.

**12** Adds a call to the `CloseFile` analysis procedure to the end of the program.

The analysis procedures used by the `prof` tool are defined in the `prof.anal.c` file as shown in the following example:

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

long *instrPerProc;
FILE *file;

void OpenFile(int n)
{
  instrPerProc = (long *) calloc(sizeof(long),n);  1
  assert(instrPerProc != NULL);
  file = fopen("prof.out","w");
  assert(file != NULL);
  fprintf(file,"%30s %15s %10s\n","Procedure","Instructions","Percentage");
}
void Count(int n, int instructions)
{
  instrTotal += instructions;
  instrPerProc[n] += instructions;
}
void Print(int n, char *name)
{
  if (instrPerProc[n] > 0) {  2
    fprintf(file,"%30s %15ld %9.3f\n", name, instrPerProc[n],
            ((float) instrPerProc[n] / instrTotal)*100.0);
  }
}
void CloseFile()  3
{
  fprintf(file,"\n%30s %15ld %9.3f\n", "Total", instrTotal,100.0);
  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.

Once the instrumentation and analysis files are specified, the tool is complete. To illustrate the application of this tool, we compile and link the "Hello" application:

```
#include <stdio.h>
main()
{
   printf("Hello world!\n");
}
```

The following example builds a call-shared executable, applies the `prof` tool, and runs the instrumented executable. In contrast to the `ptrace` tool described in Section 9.2.7.1, the `prof` tool sends its output to a file instead of `stdout`.

```
% cc hello.c -o hello
% atom hello prof.inst.c prof.anal.c -o hello.prof
% setenv LD_LIBRARY_PATH ‘pwd‘
% hello.prof
Hello world!
% more prof.out
                   Procedure    Instructions Percentage
                     __start             159      4.941
                        main              14      0.435
                           .
                           .
                           .
          _call_add_gp_range              41      1.274
       _call_remove_gp_range              35      1.088

                       Total            3218    100.000
% unsetenv LD_LIBRARY_PATH
```

### 9.2.7.3 Data Cache Simulation Tool

Instruction and data address tracing has been used for many years as a technique for capturing and analyzing 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 10GB 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 8KB 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);
}
```

Once the instrumentation and analysis files are specified, the tool is complete. To illustrate the application of this tool, we compile and link the "Hello" application:

```
#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
```

# Optimizing Techniques 10

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 Digital 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, use the `lint` command with the `–Q` flag or compile your program using the C compiler's `–check` flag (in combination with the `–migrate` or `–newc` flags) to 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 preprocessing and compilation optimizations
- 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

The following sections provide details that relate to these two aspects of the optimization process.

## 10.1 Guidelines for Building an Application Program

Opportunities for improving 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.

See Appendix D for additional optimization information that pertains only to the –oldc version of the C compiler. Appendix D contains information on uopt, the global optimizer (which is not used by the –migrate or –newc versions of the C compiler).

### 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. For details, see cc(1) or Chapter 2.

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 following compilation flags:

- For the –newc and –migrate versions of the C compiler, use –ifo and one of the following optimization-level flags:
  - When compiling with the –newc flag, use –O3 or –O4.
  - When compiling with the –migrate flag, use –O4 (preferred) or –O5.

  (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.)
- For the –oldc version of the C compiler, use –O3.

See cc(1) or Chapter 2 for information on when to use which version of the C compiler.

Note that 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 flag 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 flag to improve performance. (The highest-level optimization flag (–O4 with –newc and –O5 with –migrate) implements byte vectorization, among other optimizations, for Alpha systems.)
- For C applications that are thoroughly debugged and that do not generate any exceptions, consider using the –speculate flag. When a program compiled with this flag 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 flag can be specified in two forms:

  ```
  –speculate all
  –speculate by_routine
  ```

  Both options result in exceptions being dismissed: the –speculate all flag dismisses exceptions generated in all compilation units of the program and the –speculate by_routine flag 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 flag 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` flag.

Use of the `–speculate all` and `–speculate by_routine` flags 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
```

Both options can be specified as follows:

```
% setenv _SPECULATE_ARGS -stats -alignmsg
```

- You can use the following compilation flags together or individually with the `–newc`, `–migrate`, and `–oldc` versions of the C compiler to improve run-time performance:

| Flag | Description |
|---|---|
| `–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 flags for increased performance.<br>For `–newc`, `–migrate`, and `–oldc` versions of the C compiler:<br>`–D_INTRINSICS`<br>`–D_INLINE_INTRINSICS`<br>`–D_FASTMATH`<br>`–float`<br>`–fp_reorder`<br>`–O3` (`–O4` for `–migrate`)<br>For only `–newc` or `–migrate` versions of the C compiler:<br>`–ansi_alias`<br>`–ansi_args`<br>`–assume trusted_short_alignment`<br>`–ifo`<br>`–readonly_strings` |

| Flag | Description |
|---|---|
| `–feedback` | Specifies the name of a previously created feedback file. Information in the file can be used by the compiler when performing optimizations. |
| `–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. |
| `–Olimit` | Specifies the maximum size, in basic blocks, of a routine that will be optimized by the global optimizer (`uopt`). (This flag can be used only with the `–oldc` flag.) |
| `–om` | Performs a variety of code optimizations for programs compiled with the `–non_shared` flag. |
| `–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. |
| `–tune` | Selects processor-specific instruction tuning for specific implementations of the Alpha architecture. |
| `–unroll` | Controls loop unrolling done by the optimizer at levels `–O2` and above. (This flag can be used only with the `–newc` or `–migrate` flags.) |

Note that using the preceding flags may cause a reduction in accuracy and adherence to standards. See `cc`(1) for details on these flags.

- For C applications, the compilation flag in effect for handling floating-point exceptions can have a significant impact on execution time:
  - **Default exception handling** (no special compilation flag)

    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 flag 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 flags 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% 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, thus decreasing your application's startup time and improving 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 Digital 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.

#### 10.1.2.1 Using the Postlink Optimizer

You perform postlink optimizations by using the `–om` flag on the `cc` command line. This flag must be used with the `–non_shared` flag and must be specified when performing the final link, for example:

```
% cc -om -non_shared prog.c
```

The 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 switches, you can remove unused `.lita` entries after optimization and then compress the `.lita` section.
- Reallocation of common symbols according to a size you determine.

When you use the `–om` flag, you get the full range of postlink optimizations. To specify a specific postlink optimization, use the `–WL` compiler flag, followed by `–om_`*`option`* , where *`option`* can be one of the following:

`compress_lita`
This option removes unused `.lita` entries after optimization, then compresses the `.lita` section.

`dead_code`
This option removes dead code (unreachable options) generated after optimizations have been applied. The `.lita` section is not compressed by this option.

`ireorg_feedback,`*`file`*
This option directs the compiler to use the pixie-produced information in *`file`*`.Counts` and *`file`*`.Addrs` to reorganize the instructions to reduce cache thrashing.

`no_inst_sched`
This option turns off instruction scheduling.

`no_align_labels`
This option turns off alignment of labels. Normally, the `–om` flag will align the targets of all branches on quadword boundaries to improve loop performance.

`Gcommon,`*`num`*
This option sets the size threshold of ‘‘common’’ symbols. Every ‘‘common’’ symbol whose size is less than or equal to *`num`* will be allocated close together.

For more information, see the cc(1) reference page.

### 10.1.3 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 Digital UNIX systems, KAP uses DECthreads 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 DEC C compilation). For information on how to use KAP on a C program, see the *KAP for C for Digital UNIX User Guide*.

  KAP is available for Digital UNIX systems as a separately orderable layered product.

- Use the cord utility (–cord option) to improve the instruction cache behavior for C applications. This utility uses data from an actual run of your application to improve your application's use of the instruction cache. To use the cord utility, you must first create a feedback file with the pixie and gprof tools. See pixie(5), prof(1), cord(1), and runcord(1) for details. Also, Chapter 8 describes how to use these tools. (If you have produced a feedback file and you are are going to compile your program with the –non_shared flag, it is better to use the feedback file with the –om flag than with the –cord flag. See

Section 10.1.2.1 for details on the `om` utility.)

- To improve compiler optimizations, try recompiling your C programs with a feedback file. The C compilers can make use of data from an actual run of the program to fine tune their optimizations. For the `–newc` and `–migrate` versions of the C compiler, the feedback information is most useful at the highest two levels of optimization (`–O3` or `–O4` for `–newc` and `–O4` or `–O5` for `–migrate`). (The `–oldc` version of the C compiler does not support the use of feedback files in its processing.) If you are compiling a program with a feedback file and with the `–non_shared` flag, it is better to use the `–prof_use_om_feedback` flag than the `–prof_use_feedback` or `–feedback` flags. (See Section 10.1.2.1 for details on the `om` utility.)

  See Section 8.11 for information on how to create and use feedback files.

### 10.1.4 Library Routine Selection

Library routine options that can affect performance include the following:

- Use the Digital Extended Math Library (DXML) for applications that perform numerically intensive operations. DXML is a collection of mathematical routines that are optimized for Alpha systems – both SMP systems and uniprocessor systems. The routines in DXML 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 DXML, applications that involve numerically intensive operations may run significantly faster on Digital UNIX systems, especially when used with KAP. DXML routines can be called explicitly from your program or, in certain cases, from KAP (that is, when KAP recognizes opportunities to use the DXML routines). You access DXML by specifying the `–ldxml` flag on the compilation command line.

  For details on DXML, see the *Digital Extended Mathematical Library for Digital UNIX Systems Reference Manual*.

  The DXML routines are written in Fortran. For information on calling

Fortran routines from a C program, see the Digital UNIX user manual for the version of Fortran that you are using (DEC Fortran or DEC Fortran 90). (Information about calling DXML 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` flag 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 details.
- Consider compiling your C programs with the `–D_INTRINSICS` and `–D_INLINE_INTRINSICS` flags; 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 profiler 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.

Digital provides several profiling tools that work for programs written in C and other languages. See Chapter 8, `atom`(1), `gprof`(1), `hiprof`(5), `pixie`(5), and `prof`(1) for more details.

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, 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. Accessing an 8- or 16-bit scalar can result in a sequence of machine instructions to access the data. A 32- or 64-bit data item can be accessed with a single, efficient machine instruction.

If performance is a critical concern, 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 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, attempt to allocate these data structures on cache line boundaries in the secondary cache. Doing so can improve your application's cache usage. See Appendix A of the *Alpha Architecture Reference Manual* for additional 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 Appendix A of the *Alpha Architecture Reference Manual* for additional 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 –align flag 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 by any version of the C compiler (–newc, –migrate, or –oldc).

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 the machine code debugging capabilities of the dbx or ladebug debugger to determine the source code locations associated with pc values.

For additional information on data alignment, see Appendix A in the *Alpha Architecture Reference Manual*. See cc(1) for details on alignment-control flags that you can specify on compilation command lines.

### 10.2.3 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` flag to treat all `char` declarations as `unsigned char`.

- If your application uses large amounts of data for a short period of time, consider allocating the data dynamically with the `malloc` function instead of declaring it statically. When you have finished using the memory, free it so it can be used for other data structures later in your program. Using this technique to reduce the total memory usage of your application can substantially increase the performance of applications running in an environment in which physical memory is a scarce resource.

  If an application uses the `malloc` function extensively, you may be able to improve the application's performance (processing speed, memory utilization, or both) by using `malloc`'s control variables to tune memory allocation. See `malloc`(3) for details.

- If your application uses local arrays whose sizes are unknown at compile time, you can gain a performance advantage by allocating them with the `alloca` function, which uses very few instructions and is very efficient. Storage allocated by the `alloca` function is automatically reclaimed when an exit is made from the routine in which the allocation is made.

  The `alloca` function allocates space on the stack, not the heap, so you must make sure that the object being allocated does not exhaust all of the free stack space. If the object does not fit in the stack, a `core` dump is issued.

  Programs that issue calls to the `alloca` function should include the `alloca.h` header file. If the header file is not included, the program will execute properly, but it will run much slower.

- 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 flag. The –xtaso flag is supported by all versions of the C compiler (–newc, –migrate, and –oldc versions). To use the flag, you must modify your source code with a C-language pragma that controls pointer size allocations. See cc(1) and Chapter 2 for details.
- 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 in order to move code from within the loop to outside the loop.
- Use local variables and avoid global variables. Declare any variable outside of 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.

You should 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, whereas 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.

Two different methods can be used 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;
    }
  ```

# Handling Exception Conditions 11

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 contains the following discussions:

- Overview of exception handling
- Raising an exception from a user program
- Writing a structured exception handler
- Writing a termination handler

## 11.1 Exception Handling Overview

On Digital 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* defines special structures and mechanisms that enable the processing of exceptional events on Digital 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 Digital 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 Digital UNIX components that support the exception handling mechanism defined in the *Calling Standard for Alpha Systems.*

### 11.1.1 C Compiler Syntax

Syntax provided by the Digital 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

Routines in the exception support library, `/usr/ccs/lib/cmplrs/cc/libexc.a`, provide 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 detailed 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 Digital 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 additional details. |

**Table 11-1: (continued)**

| 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 additional details. |
| 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 additional details. |
| 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*. See pdsc(4) for additional details. |

## 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.

Examples in Section 11.3 illustrate 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 Digital UNIX C compiler allow you to deal with the possibility that a certain exception condition may occur in a certain code sequence. 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 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 additional details.

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, you should 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 simply 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 <signal.h>
#include <excpt.h>
#include <machine/fpu.h>
#include <errno.h>

main ()
{
Exception_info_ptr  except_info;
PCONTEXT            context_record;
system_exrec_type   *exception_record;
long                code;
sigset_t            newmask, oldmask;
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.
*/
    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");
```

### Example 11-1: (continued)

```
    exit(1);
}
```

The following is a sample run of this program:

```
% cc 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 <signal.h>
#include <excpt.h>
#include <machine/fpu.h>
#include <errno.h>

main ()
{
Exception_info_ptr  except_info;
PCONTEXT            context_record;
system_exrec_type   exception_record;
long                code;
sigset_t            newmask, oldmask;
struct sigaction    act, oldact;
unsigned long       float_traps=IEEE_TRAP_ENABLE_DZE, trap_mask;
int                 fpsigstate;
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:");
    if (ieee_set_fp_control(float_traps) < 0)
        {
          printf("set_fp_control problem");
          exit(1);
        }
/*
```

**Example 11-2: (continued)**

```
   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 hand-
   ler. If the filter’s test of the code is false, the filter
   returns -1 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:

          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_FAULT).
*/
          printf("No. of parameters is %u\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_FAULT)
```

### Example 11-2: (continued)

```
           {
                *((unsigned 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");
    exit(1);
}
```

The following is a sample run of this program:

```
% cc -ieee_with_no_inexact 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 0x10
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 <strings.h>
#include <stdio.h>
#define EXC_NOTWIDGET EXC_VALUE(EXC_C_USER, 1)
#define EXC_NOTDECWIDGET EXC_VALUE(EXC_C_USER, 2)
void  getwidgetbyname();
/*
   main() sets up an exception handler to field the EXC_NOTWIDGET
   exception and then calls getwidgetbyname().
```

### Example 11-3: (continued)

```
*/
main(argc, argv)
    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
   generate either of the user-defined exceptions.
*/
void
getwidgetbyname(char* widgetname[20])
{
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 DEC-supplied widget\n",
              code, widget);
```

**Example 11-3: (continued)**

```
      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 DEC-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 clean-up 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;**

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). Use of 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 illustrates 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 <signal.h>
#include <excpt.h>
#include <errno.h>

#define EXC_FOO  EXC_VALUE(EXC_C_USER, 1)

signed
foo_except_filter()
{
  printf("2. The exception causes the exception filter
             to be evaluated.\n");
  return(1);
}

main ()
{
  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_ex.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.
```

# Developing Thread-safe Libraries 12

To support the development of multithreaded applications, the Digital UNIX operating system provides DECthreads, Digital’s Multithreading Run-Time Library. The DECthreads interface is Digital UNIX’s implementation of IEEE Standard 1003.1c-1995 threads (also referred to as POSIX 1003.1c threads).

In addition to an actual threading interface, the operating system also provides Thread-Independent Services (TIS). The TIS routines are an aid to creating thread-safe libraries (see Section 12.4.1).

This chapter addresses the following topics:

- Overview of multithread support in Digital 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 Digital UNIX system include the following:

- Compiler support – Compile using the `–pthread` flag on the `cc` or `c89` command.
- Threads package – The `libpthread.so` library provides interfaces for threads control, buffers an application from lower-level threads implementation, and is selected at application link time.
- Thread-safe support libraries – These libraries include `libm.{a,so}`, `libsys5_r.a`, and `libmach.{a,so}`.
- The `ladebug` debugger
- The `prof` and `gprof` profilers – Compile with the `–p` and `–pthread` flags for `prof` and with the `–pg` and `–pthread` flags for `gprof` to use the `libprof1_r.a` profiling library.
- The `atom` utility (`pixie`, `third`, and `hiprof` tools)

For information on profiling multithreaded applications, see Section 8.14.

## 12.2 Run-Time Library Changes for POSIX Conformance

For releases of the DEC OSF/1 operating 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). The Digital UNIX operating system fixes the problem of static data in the non-reentrant versions of the routines by replacing the static data with thread-specific data. Except for a few routines specified by POSIX 1003.1c, all of the alternate routines are no longer required 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:

```
alctime_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). The new versions of the routines are the default when compiling code under Digital UNIX Version 4.0 or later, but you must be certain to include the header files specified on the manpages for the various routines.

For more information on programming with threads, see the *Guide to DECthreads* 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 lock on mutexes. (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 to be 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 Thread Independent Services (TIS) (see Section 12.4.1); this enables them to work in both single- 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 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 argument. The argument is a user-supplied buffer that is allocated by the caller. The ctime_r() function writes the following string into the buffer:

  ```
  char *ctime_r(const time_t *timer, char *buf);
  ```

  The users of this function must ensure that the buffer they supply as an argument to this function 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 pseudo-random number generator. For any given starting ‘‘seed’’ value that is set with the srand() function, it generates an identical sequence of pseudo-random 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 or keys is one way of doing this (see Section 12.4.2).

- **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 except for the creation of a potential for blocking (where the potential may not have existed previously).

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, it is good practice to declare static read-only data with the `const` type modifier to reduce the potential for misuse of the data.
- 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` flag (`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 including the `<errno.h>` file.
- Dependencies on any other nonthread-safe libraries or object files should not exist in the code.

### 12.4.1 Using Thread Independent Services (TIS)

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 DECthreads 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 DECthreads* and `tis`(3) for information on how to use this facility.

### 12.4.2 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)
{
     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);
```

**Example 12-1: (continued)**

```
          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 of ensuring 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 a limited resource. A library that needs to create a large number of data keys should instead be written to create just one and to store all of the separate data items as a structure or an array of pointers pointed to by a single key.

### 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, 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 lock 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.

### 12.5.1 Compiling Multithreaded C Applications

Many system `include` files behave differently when they are being included into the compilation of a multithreaded application. Whether the single-threaded or thread-safe `include` file behavior applies is determined by whether the ` REENTRANT` preprocessor symbol is defined. When the `–pthread` flag is supplied to the `cc` or `c89` command, the ` REENTRANT` symbol is defined automatically; it is also defined if the `pthreads.h` system `include` file is included. This `include` file must be the first file included in any application that uses the pthreads library, `libpthread.so`.

The `–pthread` flag has no other effect on the compilation of C programs. The reentrancy of the actual code generated by the C compiler is determined only by proper use of reentrant coding practices by the programmer, by use of only thread-safe support libraries, and by use of only thread-safe support libraries – not by 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` flag. When linking, the `–pthread` flag has the effect of modifying the library search path in the following ways:

- The pthreads library is included into the link.
- The exceptions and mach C libraries are included into the link.
- For each library mentioned in a `–l` flag, an attempt is made to locate and presearch a library whose name is derived by appending an ` r` to the given name.

The `–pthread` flag does not modify the behavior of the linker in any other way. The reentrancy of the linked code is determined by use of proper programming practices in the orginal code, and by compiling and linking with the proper `include` 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, you should consult the manual for the compiler you are using.

# Using 32-Bit Pointers on Digital UNIX Systems A

The Digital UNIX C compiler supports the use of 32-bit pointers on the 64-bit Digital UNIX operating system. All system interfaces use 64-bit pointers. The 32-bit pointer data type is provided to help developers reduce the amount of memory used by dynamically allocated pointers and to assist with the porting of applications that contain assumptions about the sizes of pointers. The use of 32-bit pointers in applications requires source code modifications and the use of compiler options.

## A.1 Pointer Definitions

The following list defines pointers described 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 Digital UNIX systems.
- Simple pointer: A pointer to a nonpointer data type, for example, `int *num_val;`.
- Compound pointer: A pointer to a pointer or a pointer to an indefinite array, for example, `char *argv[]` or `char **FontList`.

## A.2 Using 32-Bit Pointers

Two `cc` flags and a set of pragmas control the usage of 32-bit pointers. The `–xtaso` compiler flag causes the compiler to respond to the `#pragma pointer_size` directives. The `–xtaso_short` compiler flag causes the compiler to allocate 32-bit pointers by default and is recognized only when used with the `–xtaso` flag.

The `cc` flags for controlling pointer size are the following:

- `–xtaso`

  Enables the use of short pointers. All pointer types default to long pointers, but short pointers can be declared through the use of the `pointer_size` pragmas.

- –xtaso_short

  Enables the use of short pointers. All pointer types default to short pointers. Long pointers can be declared through the use of the pointer_size pragmas. Because all system routines continue to use 64-bit pointers, most applications require source changes when used in this way.

Within a C program, the size of pointer types can be controlled by the use of pragmas. These pragmas are only recognized by the compiler if the –xtaso or –xtaso_short flags have been specified with the cc command; they are silently ignored if neither of the flags are specified. Pointer sizes specified by the following pragmas override the default pointer size.

The #pragma pointer_size *specifier* directive provides control over pointer size allocation. This pragma has the following syntax:

**#pragma pointer_size** *specifier*

The *specifier* argument must be one of the following keywords:

long All pointer sizes following this pragma are long pointers (64 bits in length) until an overriding pointer_size pragma is encountered.

short All pointer sizes following this pragma are short pointers (32 bits in length) until an overriding pointer_size pragma is encountered.

save Save the current pointer size such that a corresponding #pragma pointer_size restore will set the pointer size to the current value. The model for pointer size preservation is a last-in, first-out stack such that a save is analogous to a push, and a restore is analogous to a pop.

restore The opposite of save. Restore the uppermost saved pointer size and delete it from the save/restore stack. 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 –xtaso flag causes the compiler to respond to the #pragma pointer_size directives. The –xtaso_short compiler flag causes the compiler to allocate 32-bit pointers by default.

The following example demonstrates the use of both short and long pointers:

```
#include <stdio.h> /* modified with #pragma pointer_size */
main ()

{
   int *a_ptr;

   printf ("A pointer is %ld bytes\n", sizeof (a_ptr));
}
```

When compiled either with default settings or with the `–xtaso` flag, the sample program prints the following:

```
A pointer is 8 bytes
```

When compiled with the `–xtaso_short` flag, this sample program prints the following:

```
A pointer is 4 bytes
```

## A.3 Syntactic Considerations

The size of pointers within macros is governed by the context in which the macro is expanded. There is no way to specify pointer size as part of a macro.

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. Thus, 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 alignment and padding rules for short pointers in structures are the same as for long pointers; the only difference is in the sizes of the pointers.

## A.4 Requirements

To use short pointers, the virtual address space in which the application runs must be constrained such that all valid pointer values are representable in 31 bits. The `–taso` linker flag enforces this constraint. Applications that use the `–xtaso` compiler flag must be linked with the `–taso` option.

## A.5 Interaction with Other Languages

Only the C compiler supports the use of short pointers. Short pointers should not be passed from C routines to routines written in other language.

## A.6 Conversion of Pointers and Other Issues

Because Digital UNIX is a 64-bit system, all applications must use 64-bit pointers wherever pointer data is exchanged with the operating system or any system-supplied libraries. Because normal applications use the standard system data types, no conversion of pointers is needed. In an application that uses short pointers, explicit conversion of the short pointers to long pointers can be required.

### A.6.1 Pointer Conversion

Conversion of pointers can be either explicit or implicit. An explicit conversion occurs when the value of a short pointer is assigned to a long pointer, or vice versa. An implicit conversion occurs when a short pointer is passed as an argument to a function that expects long pointers, or vice versa. Implicit conversions only work correctly on simple pointers; complex pointers (pointers to pointers) require explicit conversions.

In general, the conversion of complex pointers requires source code changes. Alignment and segmentation faults result if complex pointers are not correctly converted.

For example, the argument vector, `argv`, is a compound long pointer, and must be declared as such. Many X11 library functions return compound long pointers; the return values for these functions must be declared correctly or erroneous behavior will result.

The `pointer_size short` pragma has no effect on the size of the second argument to `main()`, traditionally called `argv`. This pragma always has a size of 8 bytes even if the pragma has been used to set other pointer sizes to 4 bytes.

### A.6.2 System Header Files

All Digital UNIX system routines operate on 64-bit pointers, so all system routine declarations must be made in the context of a `#pragma pointer_size long` declaration.

You can avoid extensive modification of existing applications by modifying all of the system header files on your Digital UNIX system by doing the following:

- Add the following lines to beginning of the header files:

  ```
  #pragma pointer_size (save)
  #pragma pointer_size (long)
  ```

- Add the following line to the end of the header files:

  ```
  #pragma pointer_size (restore)
  ```

The following example scripts modify the system header files to declare correctly all system routines that use long pointers. Before using these scripts, be sure to back up your system disk.

To use these scripts, create the following files in one directory and change their permissions to `execute`. Then run the `xtaso_header_edit` script with no arguments; it is automated and will modify all header files. You must be superuser on the system on which you want to perform the modifications.

```
xtaso_header_edit:
------------------
#!/bin/csh
find /usr/include ! -type l -name ’*.h’ \
     -exec short_pointer-sed.csh {} \;
find /sys/include ! -type l -name ’*.h’ \
     -exec short_pointer-sed.csh {} \;

short_pointer-sed.csh:
---------------------
#!/bin/csh
echo $1
sed -f short_ptr.sed $1 >/tmp/short_ptr.tmp
mv /tmp/short_ptr.tmp $1

short_ptr.sed:
-------------
1i \
#pragma pointer_size save
1i \
#pragma pointer_size long
$a \
#pragma pointer_size restore
```

## A.7 Restrictions

Because most applications on Digital UNIX systems use addresses that are not representable in 32 bits, the use of a short pointer in these applications would cause these applications to fail. Thus, no library that might be called by normal applications can contain short pointers. Vendors of software libraries generally should not use short pointers.

Because the use of short pointers, in general, requires understanding and knowledge of the application they are applied to, they are not recommended as a porting aid. Applications for which you are considering the use of short pointers should be ported to Digital UNIX first and then analyzed to see if short pointers would be of benefit.

The –taso linker option that is required to link programs that make use of short pointers imposes additional restrictions on the run-time environment and how libraries may be used.  See cc(1) for more information on the –taso option.

# Differences in the System V Habitat B

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:

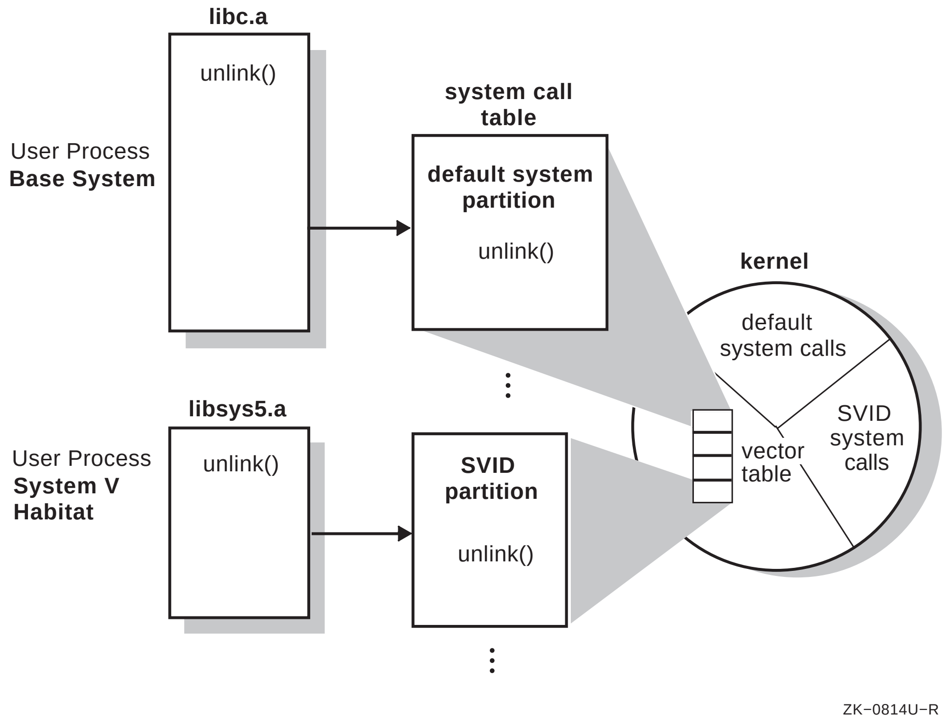
- The first level results from the modified `PATH` environment variable causing the System V versions of several user commands to execute instead of the default system versions.
- The second level results from executing the System V `cc` or `ld` commands.

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 Figure B-1 illustrates, 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.<br>**Note**<br>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.<br>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. |

# Dynamically Configurable Kernel Subsystems C

Before the Digital 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.

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
- A conceptual description of the attribute table, including example attribute tables
- An explanation of creating a configuration routine, including an example configuration routine
- A description of checking the operating system version number to ensure that the subsystem is compatible with it
- Instructions for building a loadable subsystem into the kernel for testing purposes
- Instructions for building a static subsystem that allows run-time attribute modification into the kernel for testing purposes
- Information about debugging a dynamically configurable subsystem

Device driver writers should note device-driver specific issues when writing loadable device drivers. For information about writing loadable device drivers, see *Writing Device Drivers: Tutorial*.

## C.1 Overview of Dynamically Configurable Subsystems

Many Digital 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.

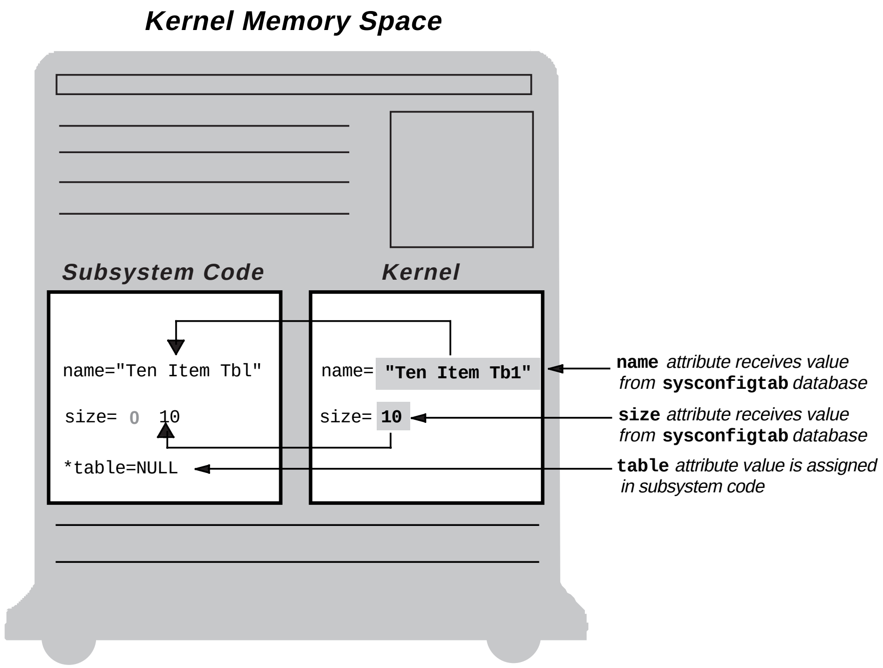
Both static and loadable subsystems can be dynamically configurable.

- 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, called 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
Subsystem Code
Kernel
name="Ten Item Tbl"
size= 0 10
*table=NULL
name= "Ten Item Tb1"
size= 10
name attribute receives value from sysconfigtab database
size attribute receives value from sysconfigtab database
table attribute value is assigned in subsystem code

ZK−0973U−R

Notice in Figure C-1 that the `size` attribute receives its initial value from the `sysconfigtab` database even though the subsystem initializes the `size` attribute to 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 declaring 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 datatypes 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 *Writing Device Drivers: Tutorial*.

The sections that follow 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 `name` field. You choose this name, which can be any string of alphabetic characters, with a length of between two characters 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 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.)

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: (continued)

```
   */
  {"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, two), 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, as listed in Table C-1.

**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. `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 sections that follow 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 reissue 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 the `/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 reissuing 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 Digital 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 != 3 && version_minor != 0)
    return EVERSION;
```

The code in this example ensures that the subsystem is running on the Version 3.0 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 issuing 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, issue 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 your 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. Shutdown 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 preceeding one in the files file, you add the subsystem to the kernel by issuing 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 preceeding 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, issue the sysconfig command with the optional –v flag. This flag 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 kernel loading software (which is called kloadsrv). Information from the kernel loading software 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 that 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 issue 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 issue 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. Issue the sysconfig command to initially configure the subsystem:

   ```
   # sysconfig -c table_mgr
   ```

3. Issue 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 issue the addobj command.)

If you want to set a breakpoint in the portion of the subsystem code that initially configures the subsystem, you must issue 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. Issue the sysconfig command to initially configure the table_mgr subsystem:

```
# sysconfig -c table_mgr
```

3. Issue 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.

# Optimizing Techniques (MIPS-Based C Compiler) D

This appendix describes the optimization phases of the `–oldc` version of the C compiler and their benefits.

## D.1 Global Optimizer

The global optimizer (`uopt`) is a single program that improves the performance of object programs by transforming existing code into more efficient coding sequences. Although the same optimizer processes optimizations for all languages, it does distinguish between the various languages to take advantage of the different language semantics involved.

The primary benefits of optimization are faster running programs and smaller object code size. However, the optimizer can also speed up development time. For example, coding time can be reduced by leaving it up to the optimizer to relate programming details to execution-time efficiency. This allows you to focus on the more crucial global structure of your program. Programs often yield optimizable code sequences regardless of how well a program is written.

## D.2 Optimizer Effects on Debugging

Optimize your programs only after they are fully developed and debugged. Although the optimizer does not alter the flow of control within a program, it may move operations so that the object code does not correspond to the source code. These changed sequences of code may create confusion when using the debugger.

## D.3 Loop Optimization by the Optimizer

Optimizations are most useful in code that contains loops. The optimizer moves loop-invariant code sequences outside loops so that they are performed only once instead of multiple times. Apart from loop-invariant code, loops often contain loop-induction expressions that can be replaced with simple increments. In programs composed of many loops, global optimization can often reduce the run time by half.

## D.4 Register Allocation by the Optimizer

Register usage has a significant impact on program performance. For example, fetching a value from a register is significantly faster than fetching a value from storage. Thus, to perform its intended function, the optimizer must make the best possible use of registers.

In allocating registers, the optimizer selects those data items that are most suited for placement in registers, taking into account their frequency of use and their location in the program structure. In addition, the optimizer assigns values to registers so that their contents move minimally within loops and during procedure invocations.

## D.5 Optimizing Separate Compilation Units

The optimizer processes one procedure at a time. Large procedures offer more opportunities for optimization because more interrelationships are exposed in terms of constructs and regions.

The `uld` and `umerge` phases of the compiler permit global optimization among separate units in the same compilation. Often, programs are divided into separate files that are compiled separately and referred to as modules or compilation units. Compiling them separately saves time during program development because a change requires recompilation of only one module, not the entire program.

Traditionally, program modularity restricted the optimization of code to a single compilation unit at a time. For example, calls to procedures that reside in other modules could not be fully optimized with the code that called them. The `uld` and `umerge` phases of the compiler system overcome this deficiency. The `uld` phase links multiple compilation units into a single compilation unit. Then, `umerge` orders the procedures for optimal processing by the global optimizer (`uopt`).

## D.6 Optimization Options

Table D-1 summarizes the functions of each of the `–O` options to the `cc -oldc` command.

**Table D-1: Compiler Optimization Options**

| Option | Result |
|---|---|
| –O3 | The uld and umerge phases process the output from the compilation phase of the compiler, which produces symbol table information and the program text in an internal format called ucode.<br>The uld phase combines all the ucode files and symbol tables, and passes control to umerge. The umerge phase reorders the ucode for optimal processing by uopt. Upon completion, umerge passes control to uopt, which performs global optimizations on the program. |
| –O2 | The uld and umerge phases are bypassed and only the global optimizer (uopt) phase executes. It performs optimization only within the bounds of individual compilation units. |
| –O1 | The uld, umerge, and uopt phases are bypassed. However, the code generator and the assembler perform basic optimizations in a more limited scope. |
| –O0 | The uld, umerge, and uopt phases are bypassed, and the assembler bypasses certain optimizations that it normally performs. |

## D.7 Full Optimization (-O3)

The following examples assume that the program prog1 consists of three files: a.c, b.c, and c.c.

To perform procedure merging optimizations –O3 on all three files, enter the following command:

```
% cc -oldc -O3 -o prog1 a.c b.c c.c
```

If you normally use the –c option to compile the object file (.o), follow these steps:

1. Compile each file separately using the –j option by entering the following commands:

   ```
   % cc -oldc -j a.c
   % cc -oldc -j b.c
   % cc -oldc -j c.c
   ```

   The –j option causes the compiler driver to produce a .u file. None of the remaining compiler phases are executed.

   The .u file contains the standard output of the first pass of the compiler (which is referred to as the front end of the compiler). The file is written in ucode, an internal language used by the compiler.

2. Enter the following command to perform optimization and complete the compilation process:

   ```
   % cc -oldc -O3 -o prog1 a.u b.u c.u
   ```

## D.8 Optimizing Large Procedures

To ensure that all procedures are optimized regardless of size, specify the `–Olimit` option at compilation time.

Because compilation time increases by the square of the procedure size, the compiler system enforces a top limit on the size of a procedure that can be optimized. This limit was set for the convenience of users who place a higher priority on the compilation turnaround time than on optimizing an entire procedure. The `–Olimit` option removes the top limit and allows those users who do not mind a long compilation to fully optimize their procedures.

## D.9 Optimizing Frequently Used Modules

You may want to optimize modules that are frequently called from other programs to reduce the compilation and optimization time required for programs calling these modules.

In the examples that follow, `b.c` and `c.c` represent two frequently used modules to be optimized, retaining all information necessary to link them with future programs; `future.c` represents one such program.

The following steps show how to optimize frequently called modules:

1. Compile `b.c` and `c.c` separately by entering the following commands:

   ```
   % cc -oldc -j b.c
   % cc -oldc -j c.c
   ```

   The `–j` option causes the front end, or first pass, of the compiler to produce two ucode files, `b.u` and `c.u`.

2. Use an editor to create a file containing the external symbols in `b.c` and `c.c` to which `future.c` will refer. The symbolic names must be separated by at least one space. Consider the following skeletal contents of `b.c` and `c.c`:

   ```
   b.c       proc1()                 c.c     x()
              {                               {
              .                               .
              .                               .
              }                               }
             proc2()                             help()
           {                                   {
   ```

```
  .                                  .
  .                                  .
  }                                  }
     proc3()                    struct
      {                          {
      .                          .
      .                          .
      }                          }  ddata;
struct                         y()
  {                            {
  .                            .
  .                            .
  }  work;                     }
```

In this example, future.c calls or references only proc1, proc2, x, ddata, and y in the two procedures (b.c and c.c). Thus, a file (named extern for this example) must be created containing the following symbolic names:

```
proc1 proc2 x ddata y
```

The structure work and the procedures help and proc3 are used internally only by b.c and c.c, and thus are not included in extern.

If you omit an external symbolic name, an error message is generated (see step 4).

3. Optimize the b.u and c.u modules using the extern file as follows:

   ```
   % cc -oldc -O3 -kp extern b.u c.u -o keep.o
   ```

   The –kp option designates that the –p linker option is to be passed to the ucode loader.

4. Create a ucode file and an optimized object code file (test_opt) for future.c, as follows:

   ```
   % cc -oldc -j future.c
   % cc -oldc -O3 future.u keep.o -o test_opt
   ```

   The following message may appear. It means that the code in future.c is using a symbol from the code in b.c or c.c that was not specified in the file extern.

   ```
   proc3: multiply defined hidden external (should have been preserved)
   ```

   If the preceding message appears, include proc3 in the file extern and recompile as follows:

   ```
   % cc -oldc -O3 -kp extern b.u c.u -o keep.o
   % cc -oldc -O3 future.u keep.o -o test_opt
   ```

## D.10 Building a ucode Object Library

Building a ucode object library is similar to building a COFF object library. First, compile the source files into ucode object files using the `–j` option:

```
% cc -oldc -j a.c
% cc -oldc -j b.c
% cc -oldc -j c.c
```

Then, enter the following commands to build a ucode library (`libtest_opt.b`) containing object files for `a.c`, `b.c`, and `c.c`:

```
% ar -crs libtest_opt.b a.u b.u c.u
```

The names of ucode libraries should have the suffix `.b`.

## D.11 Using ucode Object Libraries

Using ucode object libraries is similar to using COFF object files. To load from a ucode library, specify the `–kl`*`x`* option to the compiler driver or the ucode loader. To load from the ucode library file created in the previous example, enter the following command:

```
% cc -oldc -O3 file1.u file2.u -kltest_opt -o output
```

Libraries are searched as they are encountered on the command line, so the order in which they are specified on the command line is important. If a library is made from both assembly and high-level language routines, the ucode object library contains code only for the high-level language routines, not all of the routines as the COFF object library does. In this case, to ensure that all modules are loaded from the proper library, you must specify both the ucode object library and the COFF object library to the ucode loader.

If the compiler driver is to perform both a ucode load step and a final load step, the object file created after the ucode load step is placed in the position of the first ucode file specified or created on the command line in the final load step.

# Index

# How to Order Additional Documentation

## Technical Support

If you need help deciding which documentation best meets your needs, call 800-DIGITAL (800-344-4825) before placing your electronic, telephone, or direct mail order.

## Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-bps modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

## Telephone and Direct Mail Orders

| Your Location | Call | Contact |
|---|---|---|
| Continental USA,<br>Alaska, or Hawaii | 800-DIGITAL | Digital Equipment Corporation<br>P.O. Box CS2008<br>Nashua, New Hampshire 03061 |
| Puerto Rico | 809-754-7575 | Local Digital subsidiary |
| Canada | 800-267-6215 | Digital Equipment of Canada<br>Attn: DECdirect Operations KAO2/2<br>P.O. Box 13000<br>100 Herzberg Road<br>Kanata, Ontario, Canada K2K 2A6 |
| International | ————— | Local Digital subsidiary or<br>approved distributor |
| Internal[a] | ————— | SSB Order Processing – NQO/V19<br>*or*<br>U. S. Software Supply Business<br>Digital Equipment Corporation<br>10 Cotton Road<br>Nashua, NH 03063-1260 |

[a] For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

# Reader's Comments

**Digital UNIX**
Programmer's Guide
AA-PS30D-TE

Digital welcomes your comments and suggestions on this manual. Your input will help us to write documentation that meets your needs. Please send your suggestions using one of the following methods:

- This postage-paid form
- Internet electronic mail: `readers_comment@zk3.dec.com`
- Fax: (603) 881-0120, Attn: UEG Publications, ZKO3-3/Y32

If you are not using this form, please be sure you include the name of the document, the page number, and the product name and version.

| **Please rate this manual:** | Excellent | Good | Fair | Poor |
|---|---|---|---|---|
| Accuracy (software works as manual says) | | | | |
| Completeness (enough information) | | | | |
| Clarity (easy to understand) | | | | |
| Organization (structure of subject matter) | | | | |
| Figures (useful) | | | | |
| Examples (useful) | | | | |
| Index (ability to find topic) | | | | |
| Usability (ability to access information quickly) | | | | |

**Please list errors you have found in this manual:**

Page Description

**Additional comments or suggestions to improve this manual:**

**What version of the software described by this manual are you using?**

Name/Title Dept.

Company Date

Mailing Address

Email Phone

Do Not Cut or Tear − Fold Here and Tape

digital™

NO POSTAGE
NECESSARY IF
MAILED IN THE
UNITED STATES

**BUSINESS REPLY MAIL**
FIRST−CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
UEG PUBLICATIONS MANAGER
ZKO3−3/Y32
110 SPIT BROOK RD
NASHUA NH 03062−9987

Do Not Cut or Tear − Fold Here

Cut on
Dotted
Line