

Tru64 UNIX

SCO UNIX to Compaq Tru64 UNIX Porting Guide

Part Number: 149P-0201A-USEN

March 2001

Operating System and Version: Compaq Tru64 UNIX Version 5.1 or higher

This book presents information about porting applications from SCO UNIX to the Compaq Tru64 UNIX Version 5.1 or higher operating system.

© 2001 Compaq Computer Corporation

COMPAQ, TruCluster, ULTRIX and the Compaq logo Registered in U.S. Patent and Trademark Office. Alpha, TruCluster, and Tru64 are trademarks of Compaq Information Technologies Group, L.P. in the United States and other countries.

Microsoft and Visual C++ are trademarks of Microsoft Corporation in the United States and other countries. Intel is a trademark of Intel Corporation. Motif, OSF/1, UNIX, and X/Open are trademarks of The Open Group in the United States and other countries. All other product names mentioned herein may be trademarks of their respective companies.

Confidential computer software. Valid license from Compaq required for possession, use, or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Compaq shall not be liable for technical or editorial errors or omissions contained herein. The information in this document is provided "as is" without warranty of any kind and is subject to change without notice. The warranties for Compaq products are set forth in the express limited warranty statements accompanying such products. Nothing herein should be construed as constituting an additional warranty.

Contents

About This Manual

1 Introduction

1.1	Porting Overview	1-1
1.2	SCO UNIX Software Version	1-1
1.3	Porting and Coding Practices	1-2
1.4	Porting Tools	1-2
1.5	Porting Services	1-2

2 Comparing SCO UNIX and Tru64 UNIX

2.1	Overview of Tru64 UNIX	2-1
2.2	Real Time Programming	2-3
2.3	Directories and Defines	2-4
2.3.1	Directory Hierarchies	2-4
2.3.2	Defines for Numeric Values	2-6
2.4	Unique Directory Features	2-6
2.4.1	Device Naming	2-6
2.4.2	Context-Dependent Symbolic Links	2-6
2.5	64-Bit Considerations	2-7
2.5.1	Language Data Types	2-8
2.5.1.1	Data Access	2-8
2.5.1.2	Data Access Synchronization	2-9
2.5.2	Pointers	2-9
2.5.3	Constants	2-10
2.5.3.1	Truncation of Longs	2-10
2.5.3.2	Bit Shifts	2-11
2.5.4	Variables	2-11
2.5.5	Structures	2-11
2.5.5.1	Size	2-11
2.5.5.2	Member Alignment	2-11
2.5.5.3	Structure Alignment	2-12
2.5.5.4	Unions	2-13
2.5.5.5	Bit Fields	2-13
2.5.6	Library Calls and Operators	2-13
2.6	File System	2-14
2.7	Endian Issues	2-14
2.8	Write-to-Memory Operations and Memory Barriers	2-14

3 The Porting Process

3.1	Porting Guidelines	3-1
3.2	Porting Procedures	3-2
3.2.1	Look for Potential Problems	3-2
3.2.1.1	Use the grep Command	3-2
3.2.1.2	Use lint -Q	3-3
3.2.2	Compile the Code and Save Compiler Messages	3-3
3.2.3	Fix Problems Identified at Compile Time	3-4

3.2.4	Fix Alignment and Padding	3-4
3.3	Porting Suggestions	3-4
3.3.1	Techniques for Finding Problems in Partially Working Code	3-4
3.3.2	Hints for dbx	3-5
3.4	Porting Assistant	3-6

4 Application Development Environment

4.1	Software	4-1
4.2	C Preprocessor	4-2
4.3	C Compiler	4-2
4.3.1	Features of the Compaq C Compiler	4-2
4.3.2	Support for Interfaces and Definitions	4-3
4.3.3	Compilation Options for Hardware Architectures	4-3
4.3.4	Compilation Options for Checking Code Portability	4-3
4.3.5	Compilation Option for 32-Bit Pointers	4-4
4.3.6	Call Conventions for Passing Arguments	4-5
4.4	Linker (ld)	4-5
4.5	C++	4-6
4.6	The lint Program Checker	4-6
4.7	Debugging Tools	4-6
4.8	System V Compatibility	4-7
4.8.1	System V Habitat	4-7
4.8.2	Extended System V Compatibility	4-7
4.8.2.1	System V Environment	4-7
4.8.2.2	Extended System V Compatibility in Tru64 UNIX Version 5.0	4-8
4.9	Other Programming Tools	4-8
4.10	The make Command	4-9
4.10.1	The makefile Search Path	4-9
4.10.2	Options for the make Command	4-10
4.10.3	Implementation of make Rules	4-10
4.10.4	Make Directives	4-10
4.10.5	Built-in Macros	4-11
4.11	Header Files	4-11
4.12	Shared Libraries Provided by Tru64 UNIX	4-12
4.12.1	Tru64 UNIX Archive Libraries	4-16
4.12.2	Using Shared Libraries	4-16

5 Porting Assistant

5.1	What Porting Assistant Does	5-1
5.1.1	Code Checking	5-1
5.1.2	Diagnostic Messages	5-2
5.1.3	HyperHelp on Porting	5-2
5.1.4	Code Correction	5-2
5.1.5	Linker Assistance	5-2
5.2	How to Use Porting Assistant	5-3
5.2.1	Getting Started	5-3
5.2.2	Performing Checks	5-3
5.2.2.1	Run the Check	5-4
5.2.2.2	Step Through the Source Code	5-4
5.2.2.3	Make the Correction	5-5
5.2.2.4	Repeat the Process	5-5

5.3	Possible Limitations	5-5
6	Porting Threaded Applications	
6.1	Porting Applications That Use POSIX Threads	6-2
6.2	Porting Applications That Use DCThreads	6-2
6.2.1	Routines with Syntax Changes	6-2
6.2.1.1	pthread_attr_getinheritsched()	6-2
6.2.1.2	pthread_attr_getstacksize()	6-3
6.2.1.3	pthread_attr_setstacksize()	6-3
6.2.1.4	pthread_cleanup_pop()	6-3
6.2.1.5	pthread_cleanup_push()	6-3
6.2.1.6	pthread_cond_init()	6-3
6.2.1.7	pthread_create()	6-4
6.2.1.8	pthread_detach()	6-4
6.2.1.9	pthread_equal()	6-4
6.2.1.10	pthread_exit()	6-4
6.2.1.11	pthread_getspecific()	6-4
6.2.1.12	pthread_join()	6-5
6.2.1.13	pthread_lock_global_np()	6-5
6.2.1.14	pthread_mutex_init()	6-5
6.2.1.15	pthread_once()	6-5
6.2.1.16	pthread_setspecific()	6-5
6.2.1.17	pthread_unlock_global_np()	6-6
6.2.2	Return Values	6-6
6.2.2.1	pthread_attr_setinheritsched()	6-6
6.2.2.2	pthread_cond_timedwait()	6-6
6.2.2.3	pthread_cond_wait()	6-7
6.2.2.4	pthread_mutex_trylock()	6-7
6.2.2.5	pthread_mutex_unlock()	6-7
6.2.3	Unchanged Routines	6-8
6.2.4	New Routines	6-8
6.2.5	Routines with No Equivalent	6-9
6.3	Porting Applications That Use UnixWare Threads	6-9
6.3.1	Porting Applications That Use Read-Write Locks	6-9
6.3.2	Features in UnixWare Threads Not in the POSIX Standard	6-9
6.3.3	Features in the POSIX Standard Not in UnixWare Threads	6-10
6.3.3.1	Attributes Objects	6-10
6.3.3.2	Thread Cancellation	6-11
6.3.3.3	Scheduling	6-12
6.3.4	Synchronizing POSIX Threads	6-13
6.3.5	UnixWare Threads Routines and POSIX Threads Routines	6-14
6.3.6	For More Information	6-14
6.4	Semaphores	6-15
6.5	Programming Notes	6-15
6.5.1	Assumptions About Deadlock Conditions	6-15
6.5.2	Memory Alignment Considerations	6-15
6.5.3	POSIX Threads Library Extensions	6-16
6.5.4	Fork Handlers	6-16
6.5.5	Thread Local Storage	6-16
6.5.6	Thread-Independent Services	6-17
6.6	Files	6-17
6.6.1	Include Files	6-17
6.6.2	Shared Libraries	6-17

6.7	Linking	6-18
6.8	Compiling	6-18
6.9	Debugging	6-18
6.9.1	Ladebug Debugger	6-18
6.9.2	Visual Threads	6-20
6.9.3	ATOM	6-20

A Transferring Data

A.1	Using Fortran to Convert Unformatted Numeric Data	A-1
A.2	Nonnative Data	A-3

B Resources on the Internet

Index

Figures

2-1	Tru64 UNIX Directory Hierarchy	2-4
2-2	Byte and Bit Ordering on Alpha Systems	2-14

Tables

2-1	Contents of the Tru64 UNIX Directories	2-5
2-2	Defines from limits.h and machlimits.h	2-6
2-3	Language Data Types	2-8
2-4	Natural Data Alignment	2-8
2-5	Values of 64-Bit Constants	2-10
2-6	Structure Alignments	2-12
4-1	Software for Developers	4-1
4-2	Call Conventions for Passing Arguments	4-5
4-3	Software for System V Environments	4-7
4-4	Tru64 UNIX Programming Tools	4-8
4-5	The makefile Search Path	4-10
4-6	Options to the make Command	4-10
4-7	Special-Function Targets	4-10
4-8	Tru64 UNIX Standard Header File Directories	4-11
4-9	Shared Libraries in /usr/shlib	4-13
4-10	Shared Libraries in /usr/shlib/X11	4-15
6-1	Mapping DCThreads to POSIX Threads Library Routines	6-2
6-2	UnixWare and POSIX Threads Library Routines	6-14
6-3	Semaphore Routines	6-15
6-4	Shared Libraries	6-17
A-1	Data Conversion Keywords	A-2

About This Manual

This manual provides information about porting applications from the SCO UNIX operating system to the Compaq Tru64™ UNIX operating system. This manual also describes differences between Tru64 UNIX Version 5.1 and SCO UNIX.

Audience

This manual is for software engineers, application developers, and system managers who are considering moving software applications from SCO UNIX to Tru64 UNIX.

The manual assumes that you have software development experience and are familiar with the UNIX operating system and the SCO UNIX operating system.

Organization

This manual is organized as follows:

<i>Chapter 1</i>	Presents the advantages of porting and summarizes the porting process.
<i>Chapter 2</i>	Gives an overview of Tru64 UNIX and compares the general features of SCO UNIX and the Tru64 UNIX. It also discusses issues to consider when porting from SCO UNIX to Tru64 UNIX.
<i>Chapter 3</i>	Discusses the porting process.
<i>Chapter 4</i>	Describes the development environment on the Tru64 UNIX system.
<i>Chapter 5</i>	Describes Porting Assistant, a development environment for programmers porting applications to Tru64 UNIX.
<i>Chapter 6</i>	Presents information on porting threaded applications.
<i>Appendix A</i>	Provides information about transferring data between big-endian and little-endian systems.
<i>Appendix B</i>	Lists porting resources available on the Internet.

Related Documents

The Tru64 UNIX *Programmer's Guide* is a must for anyone writing or porting applications to Tru64 UNIX. Other books of particular interest in the Tru64 UNIX documentation set include:

- *Guide to the POSIX Threads Library*
- *Guide to Realtime Programming*
- *Compaq C Language Reference Manual*
- *Compaq Portable Mathematics Library*
- *Programmer's Guide: STREAMS*
- *Programming Support Tools*
- *Calling Standard for Alpha Systems*
- *Ladebug Debugger Manual*

For more information, see the following technical publications:

- Tru64 UNIX documentation set
- Tru64 UNIX *Software Product Description*
- SCO UNIX documentation set

Information on the Internet

You can access Tru64 UNIX documentation and other information on Tru64 UNIX on the Internet at the following location:

http://www.tru64unix.compaq.com/faqs/publications/pub_page/pubs_page.html

The Tru64 UNIX *Software Product Description* is at the following location:

<http://www.tru64unix.compaq.com/unix/technical.htm>

See *Appendix B* for a more complete list of Internet resources for Tru64 UNIX and SCO UNIX.

Reader's Comments

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

You can send your comments in the following ways:

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

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

`/usr/doc/readers_comment.txt`

Please include the following information along with your comments:

- The full title of the 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 Tru64 UNIX that you are using.
- If known, the type of processor that is running the Tru64 UNIX software.

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

Conventions

This manual uses the following typographical conventions:

#	A number sign represents the superuser prompt.
% cat	Boldface type in interactive examples indicates typed user input.
<i>file</i>	Italic (slanted) type indicates variable values, placeholders, and function argument names.
:	A vertical ellipsis indicates that a portion of an example that would normally be present is not shown.

`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.

Introduction

Porting does not have to be painful. The process of porting programs from one operating system to another inevitably involves some debugging and recompiling. But programs that adhere to language definitions, avoid nonstandard extensions, and have a minimum of architectural dependencies can run on the Tru64 UNIX operating system with few modifications.

Migration Software Systems, Ltd. of San Jose, California, has ported millions of lines of code to the Tru64 UNIX operating system on the Alpha™ architecture. Their comment on the process is instructive: "Porting to [Tru64 UNIX] can be done with a minimal amount of effort if that code was developed with proper software engineering practices."

This view is echoed in a white paper from the consulting firm D. H. Brown Associates, Inc. In *The Uneven Migration to 64-bit UNIX*, they write, "Yet, a well structured C or C++ application can be readily ported if it adheres to a few basic principles. These principles mirror generic tenets of good programming style."

When a program does require modifications, it is usually a straightforward process to identify and correct the problems. Almost all such effort involves making the code comply more closely with industry standards.

Code ported to the Tru64 UNIX operating system on the Alpha architecture enjoys several advantages:

- The ported code will run in the mature 64-bit environment that Tru64 UNIX has been providing for more than half a decade.
- The ported code will run on Alpha systems, which offer the world's fastest processors.
- The ported code will adhere closely to industry standards, making it more portable to other architectures and easier to maintain.

1.1 Porting Overview

The porting process is reasonably simple:

1. Clean up code, removing architectural dependencies and nonstandard practices.
2. Compile code. Fix problems found at compile time.
3. Fix segment faults and unaligned accesses. Unaligned accesses usually indicate a problem in the code.
4. Recompile the code and repeat the process, if necessary.

1.2 SCO UNIX Software Version

The information in this book is based on SCO UnixWare Version 7 and earlier versions of SCO UNIX. Even if the application you are porting is based on a newer version of SCO UNIX, most of the content of this book still applies.

1.3 Porting and Coding Practices

Most of the coding practices in this book are not specific to any one vendor, nor are they specific to the Tru64 UNIX operating system. They are simply good, standard coding practices. A number of books are available that cover similar ground. Some of the good ones (but certainly not the only good ones) include:

- *The C Programming Language, Second Edition*, by Brian W. Kernighan and Dennis M. Ritchie; Prentice-Hall Software Series, Prentice-Hall, Inc.
- *C Traps and Pit Falls*, by Andrew Koenig; Addison-Wesley Publishing Company

1.4 Porting Tools

Compaq Porting Assistant is a set of integrated software tools designed to help port applications to Tru64 UNIX. It checks for things that are likely to cause porting problems, and it simplifies the process of making needed changes.

See Chapter 5 for more information.

1.5 Porting Services

If you do not want to take on the task of porting your application yourself, the following resources can help you get the job done:

- The Compaq Service Provider Program offers a comprehensive portfolio of services. You can find information about the program at the following location:

<http://www.compaq.com/enterprise/sp/solutions-centers.html>

The Compaq Solutions Alliance (CSA) program provides services that can be of significant help in migrating software and in using Tru64 UNIX. You can reach them at:

1-800-332-4786

alpha-developer@compaq.com

<http://csa.compaq.com>

- Third-party consulting sources

Companies such as Migration Software Systems, Ltd. of San Jose, California, can help you to port your existing applications and systems to the Tru64 UNIX environment. Contact Migration Software Systems at 408-452-0527, or visit the following web site:

<http://www.migration.com/home.html>

Comparing SCO UNIX and Tru64 UNIX

This chapter compares major features of SCO UNIX and Tru64 UNIX. It also describes features available only on Tru64 UNIX. It presents porting issues involving the Alpha architecture and the 64-bit environment. If you are already familiar with Tru64 UNIX, you can skip this material.

This chapter presents the following topics:

- An overview of Tru64 UNIX
- Real Time Programming
- Directories and Defines
- Unique Directory Features
- 64-bit considerations
- File systems
- Endian issues
- Write-to-memory operations and memory barriers

2.1 Overview of Tru64 UNIX

The Tru64 UNIX operating system is a 64-bit advanced kernel architecture based on Carnegie-Mellon University's Mach V2.5 kernel design, with components from Berkeley Software Distribution (BSD) 4.3 and 4.4, UNIX System V, and other sources. Tru64 UNIX implements the Open Software Foundation (OSF) OSF/1 R1.0, R1.1, and R1.2 technology, Common Desktop Environment (CDE), and the Motif graphical user interface and programming environment. Under the X/Open UNIX branding program, Compaq Computer Corporation has received the UNIX 95 brand for the Tru64 UNIX operating system.

Tru64 UNIX provides symmetric multiprocessing (SMP), realtime support, and numerous features to assist programmers in developing applications that use shared libraries, multithread support, and memory-mapped files. All features of the X Window System, Version 11, Release 6 (X11R6) from the X Consortium, Inc. are fully supported. Selected features of Release 6.1 (X11R6.1) are also supported. POSIX standard 1003.1c pthreads are supported.

The Tru64 UNIX operating system complies with numerous other standards and industry specifications, including the X/Open XPG4 and XTI, POSIX, FIPS, and System V Interface Definition (SVID). Tru64 UNIX is compatible with the Berkeley 4.3 and System V programming interfaces and conforms with the OSF Application Environment Specification (AES), which specifies an interface for developing portable applications that run on a variety of hardware platforms.

Familiar User Interfaces and Tools

Tru64 UNIX provides user interfaces that are familiar to UNIX developers:

- GUI:
 - Common Desktop Environment (CDE)
 - X11R6
 - Motif 1.2

- Shells:
 - POSIX shell
 - C shell (csh)
 - Bourne shell
 - Korn shell (ksh)
- Editors:
 - vi
 - GNU emacs
 - ed
 - ex
- Development Tools

The Developers' Toolkit for Tru64 UNIX provides the following tools:

- Compaq C compiler for Tru64 UNIX

The Compaq C compiler is NIST-validated, conforms to the ANSI X3J11/88-159 C Language Standard (equivalent to ANSI X3.159-1989 and ISO/IEC 9899:1990), and provides IEEE floating-point support conformant with ANSI/IEEE Standard 754. It provides support for ANSI, strict ANSI, and Kernighan and Ritchie (K&R)-style programming.
- C macro preprocessor, cpp
- Standard C libraries
- dbx debugger
- Ladebug debugger
- Porting Assistant
- Visual Threads
- Program analysis tools
- Procedure reordering tools
- ATOM API

The following are separately packaged products:

- Compaq C++
- Compaq Visual Fortran
- Compaq Pascal
- Compaq COBOL
- Compaq Ada
- Compaq Portable Math Library (CPML)
- Compaq KAP preprocessors
- Compaq FUSE
- Compaq Enterprise Toolkit
- Compaq Open3D
- Java
 - Java Development Kit for Tru64 UNIX
 - Java 2 SDK (J2SDK) for Tru64 UNIX
 - Just-in-Time (JIT) compiler

- Java 2 Run-time Environment (JRE) for Tru64 UNIX
 - Compaq Fast Virtual Machine (Fast VM) for Tru64 UNIX
 - Compaq JTrek
 - Compaq Servit
 - Compaq AttachLayout
 - Networking
- Tru64 UNIX supports a wide range of industry-standard networking components including:

ASU	Fast Ethernet	Packetfilter
ATM	FDDI	screend
BIND	Gigabit Ethernet	Slow Ethernet
BSD Sockets	IP Multicast	SNMP
DCE	LAN Manager	SysV R4 STREAMS
DECnet	LAT	TCP/IP
DLB	LAT/Telnet gateway	Token Ring
DLI	NTP	TSP
DNS	ONC	XTI/TLI

- Data Interoperability

Support for common data formats provides data interoperability:

- NFS V2 and V3
- Interchangeable tar formats
- ISO 9600 CD-ROM file system

For more information about Tru64 UNIX features, see the Tru64 UNIX *Technical Overview* and the Tru64 UNIX *Software Product Description*.

The *Technical Overview* is at the following location:

http://www.tru64unix.compaq.com/faqs/publications/pub_page/pubs_page.html

The Tru64 UNIX *Software Product Description* is at the following location:

<http://www.tru64unix.compaq.com/unix/technical.htm>

2.2 Real Time Programming

The Tru64 UNIX operating system supports facilities to enhance the performance of realtime applications. These realtime facilities make it possible for the operating system to guarantee that the realtime application has access to resources whenever it needs them and for as long as it needs them. That is, the realtime applications running on the operating system can respond to external events regardless of the impact on other executing tasks or processes. Realtime applications written to run on the operating system make use of and rely on the following system capabilities:

- A preemptive kernel
- Fixed-priority scheduling policies
- Real-time clocks and timers
- Memory locking
- Asynchronous I/O

- File synchronization
- Queued realtime signals
- Process communication facilities

All of these realtime facilities work together to form the Tru64 UNIX realtime environment. In addition, realtime applications make full use of process synchronization techniques and facilities.

For more information on the real-time programming environment, see the *Guide to Realtime Programming*. For information on configuring the real-time kernel, see the *System Administration* guide.

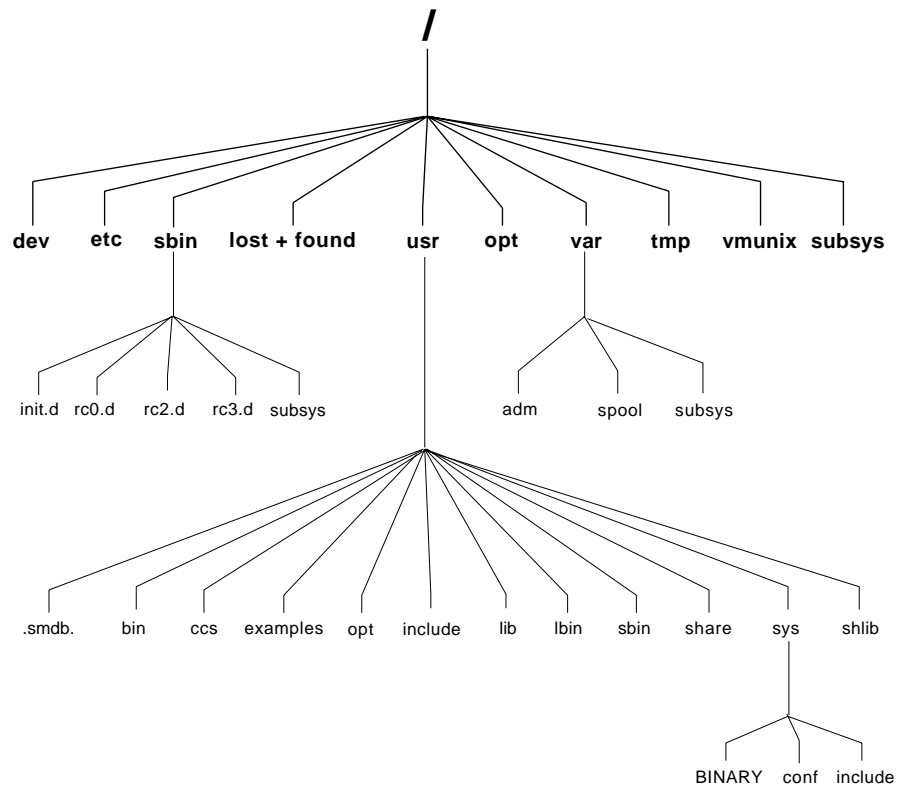
2.3 Directories and Defines

Among the differences between SCO UNIX and the Tru64 UNIX operating system are the layout of the directories and the values associated with some of the standard defines.

2.3.1 Directory Hierarchies

The hierarchies of system directories differ between SCO UNIX and Tru64 UNIX. Figure 2–1 shows the Tru64 UNIX directory structure.

Figure 2–1: Tru64 UNIX Directory Hierarchy



ZK-1369U-AI

Table 2–1 describes the contents of the Tru64 UNIX directories.

Table 2–1: Contents of the Tru64 UNIX Directories

Directory	Description
/	The root directory of the file system.
/dev	Block and character device files.
/etc	System configuration files and databases; nonexecutable files.
/sbin	Commands essential to boot the system; these commands do not depend on shared libraries or on the loader and can have other versions in /usr/bin or /usr/sbin.
/sbin/init.d	System initialization files.
/sbin/rc0.d	The run control files executed for system-state 0 (single-user state).
/sbin/rc2.d	The run control files executed for system-state 2 (nonnetworked multiuser state).
/sbin/rc3.d	The run control files executed for system-state 3 (networked multiuser state).
/sbin/subsys	Loadable kernel modules required in single-user mode.
/lost+found	Files recovered by fsck.
/usr	Most user utilities and applications. Most commands in /usr/bin, /usr/sbin, and /usr/sbin have been built with the shared version of libc and will not work unless /usr is mounted.
/usr/.smbd.	Subset installation control files used by setld.
/usr/bin	Common utilities and applications.
/usr/ccs	C compilation system; tools and libraries used to generate C programs.
/usr/examples	Source code for example programs.
/usr/opt	Optional application packages, such as layered products.
/usr/include	Program header (include) files. For more information, see Chapter 4, Application Development Environment.
/usr/lib	Libraries, data files, and symbolic links to library files located elsewhere; included for compatibility.
/usr/lbin	Back-end executable files.
/usr/sbin	System administration utilities and system utilities.
/usr/share	Architecture-independent ASCII text files. This directory includes word lists, various libraries, and online reference pages.
/usr/sys	Directories that contain system configuration files.
/usr/shlib	Binary loadable shared libraries; shared versions of libraries in /usr/ccs/lib.
/opt	Optional application packages, such as layered products.

Table 2–1: Contents of the Tru64 UNIX Directories (cont.)

Directory	Description
<code>/var</code>	Multipurpose log, temporary, transient, varying, and spool files.
<code>/var/adm</code>	Common administrative files and databases. This directory includes the crash area, files for the <code>cron</code> daemon, configuration and database files for <code>sendmail</code> , and files generated by <code>syslog</code> .
<code>/var/spool</code>	Miscellaneous printer and mail system spooling directories.
<code>/tmp</code>	System-generated temporary files that are usually not preserved across a system reboot.
<code>/vmunix</code>	Pure kernel executable file (the operating system loaded into memory at boot time).
<code>/genvmunix</code>	Generic kernel executable file built with most options and device support (useful if <code>vmunix</code> becomes corrupted).

2.3.2 Defines for Numeric Values

Some maximum and minimum values assigned to standard defines vary according to system architecture. Table 2–2 shows values for defines in `/usr/include/alpha/machlimits.h` (Tru64 UNIX) and `/usr/include/limits.h` (UnixWare and Tru64 UNIX). A number of defines in the UnixWare `/limits.h` file do not appear in the Tru64 UNIX `limits.h` file.

Table 2–2: Defines from limits.h and machlimits.h

Define	UnixWare	Tru64 UNIX
<code>LONG_BIT</code>	32	64
<code>MB_LEN_MAX</code>	5	4

2.4 Unique Directory Features

There are two unique directory features in Tru64 UNIX:

- Device naming
- Context-dependent symbolic links

2.4.1 Device Naming

Device special files appear in a `/devices` directory under the root directory (`/`). This directory contains subdirectories each containing device special files for a class of devices. A class of device corresponds to related types of devices, such as disks or nonrewind tapes. For example, the directory `/dev/disk` contains files for all supported disks, and `/dev/ntape` contains device special files for nonrewind tape devices. Currently, only the subdirectories for certain classes have been created.

For more information, see the section Device Naming and Device Special Files in the *System Administration* manual.

2.4.2 Context-Dependent Symbolic Links

The cluster environment provided by the TruCluster™ Server software has a single, clusterwide namespace for files and directories. This namespace gives each cluster member the same view of all file systems. All cluster members use

the same file name to access a given file regardless of where the file actually resides in the cluster.

Some configuration files and directories cannot be shared by all cluster members. For example, a member's `/etc/sysconfigtab` file contains information specific to that member's kernel component configuration, and only that member should use that configuration. Context-dependent symbolic links (CDSLs) provide a mechanism that lets each member read and write the file named `/etc/sysconfigtab`, while actually reading and writing its own member-specific `sysconfigtab` file.

A CDSL is a special kind of symbolic link that contains a variable whose value is determined only during pathname resolution. The `{memb}` variable is used to access member-specific files in a cluster. The following example shows the CDSL for `/etc/sysconfigtab`:

```
/etc/sysconfigtab -> ../cluster/members/{memb}/etc/sysconfigtab
```

When resolving a CDSL pathname, the kernel replaces the `{memb}` variable with the string member `n`, where `n` is the member ID of the current member. Therefore, on a cluster member whose member ID is 2, the pathname `/cluster/members/{memb}/etc/sysconfigtab` resolves to `/cluster/members/member2/etc/sysconfigtab`.

CDSLs also appear in standalone Tru64 UNIX systems, but are not used in the standalone environment.

For more information, see the section Context-Dependent Symbolic Links and Clusters in the *System Administration* manual.

2.5 64-Bit Considerations

The Alpha architecture is based on a 64-bit microprocessor, and Tru64 UNIX is a 64-bit operating system. These facts introduce a number of extended capabilities beyond 32-bit architectures. For example, 64-bit addressing allows Tru64 UNIX to support file system sizes greater than 2 gigabytes.

When porting a 32-bit application to the 64-bit environment of Tru64 UNIX, you face the same issues you would in porting that application to 64-bit SCO UNIX. This section describes these issues.

UnixWare Version 7 follows the LP64 model, the industry programming model for 64-bit UNIX. LP64 is based on the Tru64 UNIX implementation in use since 1993. If your application is written for a 64-bit environment and compiles in 64-bit mode on SCO UNIX (`-xarch=v9`), then the only 64-bit considerations in porting the application to Tru64 UNIX might be issues of data and structure alignments, covered later in this section.

Most porting concerns for applications written for a 32-bit environment are caused by three facts about the 64-bit environment:

- Pointers are 64 bits, not 32 bits.
- Long values (`long`) are 64 bits, not 32 bits.
- Integer values (`int`) are 32 bits and not the same size as longs and pointers, as is the case in a 32-bit environment.

Chances are that most code you end up changing incorrectly assumes the following:

```
sizeof(int) == sizeof(pointer) == sizeof(long)
```

2.5.1 Language Data Types

In the Tru64 UNIX 64-bit environment, `long` and `pointer` data types are 64 bits. Table 2-3 presents the Tru64 UNIX data types.

Table 2-3: Language Data Types

Data Type	Tru64 UNIX Bits
<code>char</code>	8
<code>short</code>	16
<code>int</code>	32
<code>float</code>	32
<code>pointer</code>	64
<code>long</code>	64
<code>long long</code>	64
<code>double</code>	64
<code>long double</code>	128

Table 2-4 shows the alignment of various Tru64 UNIX data types. Note that `long` and `pointer` data types are aligned on 8-byte boundaries.

Table 2-4: Natural Data Alignment

Data Type	Alignment (Byte Multiple)
<code>char</code>	4
<code>short</code>	4
<code>int</code>	4
<code>float</code>	4
<code>long</code>	8
<code>pointer</code>	8
<code>long long</code>	8
<code>double</code>	8

With high-level languages, such as C, the compiler automatically attempts to align data and variables to their natural boundaries. In some situations, the compiler might lack the information needed to make the correct alignment. Alignment errors can result from misuse of `long` and `pointer` data types in structure definitions that are shared between 32-bit and 64-bit systems.

2.5.1.1 Data Access

Older Alpha processors (EV4 and EV5) only support memory access of longwords (32 bits) and quadwords (64 bits). Byte and word accesses are accomplished by multiple instructions that load a longword or quadword, mask, and shift to obtain the desired entity. The lack of a single, uninterruptible operation for byte and word access has implications both for the performance and the correctness of an application.

Beginning with the 21164a Alpha chip (EV56), the Alpha processor supports direct access of byte and word data types, as well as direct access of longword and quadword data types.

In Tru64 UNIX Version 4.0 and higher, the operating system kernel includes an instruction emulator that allows any Alpha chip to execute and produce

correct results from Alpha instructions, even if some of the instructions are not implemented on the chip. Applications that use emulated instructions will run correctly, but might incur significant emulation overhead at run time.

To learn the type of Alpha processor on your system, use the following command:

```
/usr/sbin/psrinfo -v
```

2.5.1.2 Data Access Synchronization

Independently executing applications that access common data must synchronize access to that data. The Alpha architecture mandates that for naturally aligned quadwords, independent access to adjacent quadwords produces the same results regardless of the order of instruction execution. No such guarantee exists for `char`, `byte`, `word`, or `longword` data.

A multithreaded application or multiple processes that access adjacent `char`, `byte`, `word`, or `longword` data through a common address space must use either thread mutex locking functions or semaphore locks to ensure that access to the data is deterministic. Similarly, such processes using shared memory-mapped files are restricted to semaphore locks to avoid conflicts with access operations to adjacent data items of type `char`, `byte`, `word`, or `longword`.

A simpler alternative to locking functions or semaphores is to expand `byte`-, `word`-, or `longword`-length items to quadwords.

Quadword data items that are not naturally aligned (low-order 3 bits) incur access penalties similar to those for `byte` and `word` access. But the compiler takes care of correct alignment for quadword data.

Compiling an application with the `-arch ev56` option reduces the chances of word tearing for `byte`-, `word`-, or `longword`-length items. Even so, there are still cases where the compiler will generate code that exposes word tearing.

The `cc -granularity <size>` option ensures that the compiler generates code that will not cause word tearing for the `<size>` specified. However, Tru64 UNIX system libraries and kernel interfaces frequently assume quadword granularity. The compiler alone cannot resolve all the issues if you require granularity of less than an 8-byte quadword.

Note

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

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

For more information on data alignment and threads, see Section 6.5.2.

2.5.2 Pointers

Pointers are 64 bits long. Treating pointers as though they are the same size as `int` values will likely cause unwanted results.

Code to be ported that casts a pointer or quadword (type `long` or `u_long`) to an `int` value results in the upper 32 bits being truncated. If that `int` value is then cast back to 64 bits, the resulting value is incorrect.

The following summarizes the behavior of 64-bit pointers:

- **Truncation of pointers**
Pointers assigned to `int` variables are truncated to 32-bit values. Similarly, pointers passed to functions expecting `int` values are truncated to 32 bits.
- **Assigning integer values to pointers**
Assigning a pointer to an `int` variable, then assigning the `int` back to the pointer, and dereferencing the pointer can result in a segmentation fault.
- **Subtraction of pointers**
In Tru64 UNIX, the length of the integer required to hold the difference between two pointers to members of the same array is a signed `long`. This value, `ptrdiff_t`, is defined in the `stddef.h` file.

2.5.3 Constants

Constants can have different values on 32-bit and 64-bit systems. Table 2-5 lists some constants and their values.

Table 2-5: Values of 64-Bit Constants

C Constant	Value	32-Bit Value	64-Bit Value
<code>0xffffffff</code>	$(2^{32} - 1)$	-1	4,294,967,295
<code>4294967296</code>	2^{32}	0	4,294,967,296
<code>0x100000000</code>	2^{32}	0	4,294,967,296
<code>0xffffffffffffffff</code>	$(2^{64} - 1)$	-1	-1

In the following code fragment, the expression in the `if` statement is true in a 32-bit environment but false in Tru64 UNIX:

```
long long_val = 0xffffffff;
if(long_val < 0)
```

In Tru64 UNIX, `long` and unsigned `long` constants are 64 bit, quadword values. For example:

```
sizeof(543210) = 4 bytes
sizeof(543210L) = 8 bytes
sizeof(543210UL) = 8 bytes
```

2.5.3.1 Truncation of Longs

Because `longs` are 64-bit values, truncation can occur if a `long` is assigned to an `int` variable. For example:

```
int int_val;
.
.
.
int_val = 2147483660;
```

Because of truncation, the value of `int_val` is -2147483636.

Truncation can also occur if a `long` value is passed as an argument to a function expecting an `int` value. For example:

```
abs(2147483660) = 2147483636
```

2.5.3.2 Bit Shifts

A bit-shift operation on an integer constant always yields a 32-bit constant. For example, even though `long_val` is declared a `long`, the results of the following operations are 32-bit values:

```
long_val = 1 << 31 results in long_val = -2147483648 or 0xffffffff80000000
if((1 << 31) > 0x7fffffff) is false
```

If you need a result of type `long`, you must use the `L` or `UL` suffix for `long` integer constants. The top 32 bits of value depend on the type of the value shifted. Signed values are sign extended; unsigned values are zero extended. If you want a 64-bit constant, be sure to use the `L` or the `UL` suffix. Only the left operand of a shift operator determines the result type. The type of shift count operand is irrelevant.

```
long_val = 1L << 31 results in 2147483648 or 0x80000000
if((1L << 31) > 0x7fffffff) is true
```

You obtain similar results by casting to a `long`. For example, when shifting bytes into a `long` value, cast each byte to a `long`; otherwise, the result is only a 32-bit value. The following example results in a 64-bit value. (Assume `long_val` is a `long` data type and `bp` is a pointer to bytes.)

```
long_val = (((u_long)bp[0] << 56) | ((u_long)bp[1] << 48));
```

2.5.4 Variables

Variables declared as `int` are 32-bit entities on both 32-bit systems and in the 64-bit environment of Tru64 UNIX. Variables declared as `long` (and as pointers) are 64 bits in Tru64 UNIX.

If you have specific variables that must be 32 bits in size on both Tru64 UNIX and 32-bit systems, define the type to be `int`. If the variable must be 32 bits on 32-bit systems but 64 bits on Tru64 UNIX systems, define the variable to be `long`.

2.5.5 Structures

The 64-bit environment can affect both the size and alignment of structures, as described in this section.

2.5.5.1 Size

Because pointers and `longs` are 64-bit values, structures and unions that include pointers or `long` data types are larger than the same structures and unions on 32-bit systems.

For example, the following structure, `TextNode`, doubles in size on a 64-bit system because the pointer types are doubled in size from 4 bytes to 8 bytes:

```
struct TextNode
{
char *text;
struct TextNode *left;
struct TextNode *right;
};
```

If you are sharing data defined in structures between 32-bit and 64-bit systems, avoid using `longs` and pointers as members in shared structures.

2.5.5.2 Member Alignment

The compiler ensures that members of structures and unions are aligned on their natural boundaries. Table 2-6 shows the alignments of various data types.

Table 2–6: Structure Alignments

Data Type	Alignment
char	byte
short	word
int	longword
long	quadword
pointer	quadword

The compiler sometimes inserts padding to provide member alignment in structures and unions. On 64-bit Alpha systems, the size of the following structure is 32 bytes: 8 bytes for each pointer and 4 bytes of padding after the `int` member size, so that the pointer `left`, which follows `size`, is aligned on a 64-bit boundary:

```
struct TextCountNode
{
char *text;
int size;
struct TextCountNode *left;
struct TextCountNode *right;
};
```

2.5.5.3 Structure Alignment

The compiler aligns structures according to the strictest aligned member. This aids in aligning structure members on their required boundaries. The compiler pads structures to ensure proper alignment. Padding can be added within the structure or at the end of the structure to terminate the structure on the same alignment boundary on which it started.

Because of padding, do not assume that the size of a structure is simply the accumulated size of all of the objects defined in it. The `sizeof` operator is a safer method for determining structure size.

In some cases, you can minimize the amount of padding needed in a structure by reordering the members.

The following structure is 40 bytes; the compiler adds 4 bytes of padding after each of the members `size` and `count`, to maintain alignment of the pointers on 64-bit boundaries:

```
struct TextCountNode
{
char *text;
int size;
struct TextCountNode *left;
int count;
struct TextCountNode *right;
};
```

Placing the two `int` members together eliminates the padding and reduces the size of the structure to 32 bytes:

```
struct TextCountNode
{
char *text;
int size;
int count;
struct TextCountNode *left;
struct TextCountNode *right;
};
```

2.5.5.4 Unions

Problems arise when the use of a union is based on assumptions such as the following:

```
sizeof(double) == 2*sizeof(long) or sizeof(long) == 4*sizeof(char)
```

The following code fragment assumes that an array of two longs overlays a double:

```
union double_union {
double d;
unsigned long ul[2];
};
```

Changing the long to an int fixes the problem:

```
union double_union {
double d;
unsigned int ul[2];
};
```

2.5.5.5 Bit Fields

Bit fields are allowed on any integral type on Alpha systems. (ANSI C requires only bit fields with int, signed int, and unsigned int types.)

In a C declaration, if one bit field immediately follows another in a structure declaration, the second bit field is packed into adjacent bits of the former unit. Because the long data type is 64 bits long on Alpha systems, consecutive declarations of bit fields of type long can contain multiple bit-field definitions, whereas this might not occur on 32-bit systems. This difference can cause unexpected results in operations on these bit fields.

To ensure the same behavior in operations on bit fields, change bit field definitions of type long to int.

The `-Zp n` option to the `cc` command and the `#pragma pack` directive let you specify the number of bytes used to align the members of a structure. For more information, see the *Tru64 UNIX Programmer's Guide* and `cc(1)`.

2.5.6 Library Calls and Operators

The 64-bit data types also affect the following library calls and operator:

- `printf()` and `scanf()`

The conversion specifications for `printf()` and `scanf()` have been modified to accommodate the fact that longs and unsigned longs are 64-bit values. Use the length modifier, `l` (lowercase letter L), with the `d`, `u`, `o`, and `x` conversion characters to specify assignment of type long or unsigned long.

For example, `%ld` prints a long as a signed decimal, and `%lu` prints a long as an unsigned decimal. If the length modifier is not used, the type is assumed to be int, or unsigned int, depending upon the conversion character.

The format code `%Lf` specifies a 128-bit long double type.

- `malloc()` and `calloc()`

Memory allocation library functions such as `malloc()` guarantee to return data aligned to the natural alignment of any object. In the 64-bit Tru64 UNIX environment, `malloc()` returns a pointer to memory that is at least quadword aligned.

- `lseek()`

The `lseek()` call takes an argument of type `long (off_t)` to specify the file offset.

- `fsetpos()` and `fgetpos()`

The `fsetpos()` and `fgetpos()` calls take an argument of type `long (fpos_t)` to specify the file position.

- `varargs()`

In Tru64 UNIX, `va_list` is not a `char` pointer. The formal declaration of `va_list` is given in the `va_list.h` file. See `varargs(3)` for more information on the use of these macros, as well as the data manipulation section in *Calling Standard for Alpha Systems*.

- `sizeof` operator

The result of the `sizeof` operator is `size_t`, an unsigned `long`.

2.6 File System

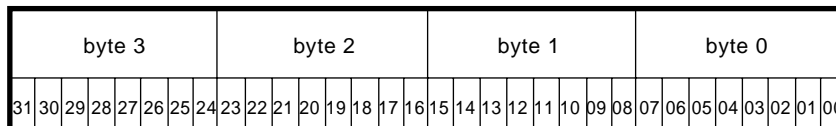
The 64-bit Tru64 UNIX operating system allows you to build very large files and file systems. The `off_t` file offset is defined to be a `long` on Alpha systems (64 bits). Given this extended capability, you can build files and file systems that cannot be fully accessed by 32-bit systems. Consider this when working in a distributed environment in which file systems are shared between 32- and 64-bit systems.

2.7 Endian Issues

Like Tru64 UNIX on Alpha, SCO UNIX on the Intel x86 architecture is little endian. Byte 0 is the least-significant byte. You do not need to take endianism into consideration when you port code from SCO UNIX to Tru64 UNIX.

Figure 2-2 illustrates the byte ordering on the Alpha architecture.

Figure 2-2: Byte and Bit Ordering on Alpha Systems



ZK-1370U-AI

For well-constructed code, the endianism of a system is almost always transparent. Those few cases in which endianism is a concern typically are caused by coding practices that mix types in unions or casts.

2.8 Write-to-Memory Operations and Memory Barriers

The Alpha architecture guarantees coherency of a processor's view of memory (that is, cache is updated, or the contents marked invalid and good data fetched elsewhere). The architecture has a shared-memory model that specifies no implicit ordering between the reads and writes issued on one processor, as viewed by a different processor. This approach allows a wide variety of high-performance implementation techniques. For example, it makes possible such implementations as the use of multibank caches, bypassed write buffers, write merging, and pipelined writes with retry on error.

When required, specify strict ordering of reads and writes by using explicit memory barrier (MB) instructions. Low-level hardware operations, such as device drivers, often make use of memory barrier instructions to ensure the order in which data are written to memory.

The following code fragment illustrates the use of a memory barrier:

```
device_intr()
{
mb();
bcopy (DMA_buffer, data, nbytes);
/* If we need to update a device register, do: */
mb();
device->csr = DONE;
mb();
}
```

See *Writing Device Drivers* in the Tru64 UNIX Device Driver Kit for more information.

The Porting Process

This chapter consists of general porting guidelines, followed by a procedure you can use to port code to the Tru64 UNIX operating system.

3.1 Porting Guidelines

The following suggestions are not the only way to port a 32-bit application to a 64-bit environment. However, each step is based on the lessons learned by people who have already ported code, so consider them before creating your own approach.

- Obtain copies of each system's commonly used `.h` files (for example, `limits.h`, `types.h`, `param.h`, and so on) and use them for reference. Use predefined types whenever possible. Use the `*io.h`, `common.h`, and `iotypes.h` files for driver and kernel typedefs.
- Take the time to create and use function prototypes. The time invested doing this will be repaid later. Be aware of function arguments that are not explicitly declared and typed. Argument sizes of such functions might not match those of the calling program.

Also note that `int` is the default size for untyped register and unsigned variables.

- Check all code with `lint`. Use the `-Q` option, which turns on checking for potential migration issues when moving from a 32-bit to a 64-bit architecture.
- When declaring constants, use `L` or `UL` as appropriate; when necessary, use the unsigned `UL` to prevent sign extension.
- When checking or converting long declarations, check calls to `printf()` and appropriately change format strings to `%ld`, `%lu`, `%lx`, or vice versa. (Most people miss these changes during their porting.)
- Check `(int *)` and `(long *)` types of casts.
- Use `NULL`, defined as `0L`, for zero or `(char *)` comparisons.
- Declare variables as `int` or `long` for alignment and performance. Do not try to save bytes by using a `char` or a `short`.
- Generate appropriate typedefs for portability. For example:

```
typedef int boolean_t;
```

- Beware of rounding pointers for alignment (for example, `& 0x03`). To avoid truncating addresses, use appropriate castings, and for size, use the following:

```
(sizeof(long) - 1)
```

For example:

```
#define round(a,b) (((ulong)(a)+(b))-1)&~(ulong)((b)-1))
```

```
.  
.
.
```

```
rndstak = (uchar_t *)round(staktop, BYTESPERWORD);
```

- Avoid storing structures that contain pointers in data files. These files then become nonportable between 32-bit and 64-bit systems.
- Declare character pointers and character bytes as unsigned `char` to avoid sign extension problems with 8-bit characters.

- The `signal()` function returns a pointer (64 bits) to the previous signal handler:

```
void (*signal(int signal, void (*function)( int ))) (int);
```

Do not store the pointer in an `int` or the address will be truncated.

- Use variable argument lists instead of declaring extra `int` arguments.

Note

The Alpha `va_list` is not a `char` pointer as it is with many other `varargs` implementations. See the `va_list.h` file for the formal declaration of `va_list`.

- When shifting bytes into a `long` value, ensure that each byte is cast to a `long`. Otherwise, the result is only a 32-bit value (not 64 bits as expected).
When shifting bits on an integer constant, specify the constant with `L` or `UL` if you want a result of type `long` or `unsigned long`, respectively. Otherwise, the results will be an integer. See Section 2.5.3.2 for more information.
- Beware of compiler optimizations. Declare as `volatile` any variable used in a loop that is updated by an external function (for example, by a signal handler). Otherwise, the loops will not detect changes.
- In driver `ioctl()` functions, ensure that the `cmd` argument is an `unsigned int` to avoid sign extension and comparison problems. Any command defined with `IOC_IN (0x80000000)` will sign extend.
- Network Internet addresses are 32 bits. Most network code uses `long`s for network addresses (16-bit leftover necessary to force 32 bits).
- When data alignment cannot change (for example, network packets), use `#pragma pack` to avoid compiler structure padding.
- Change only what you know is broken. This approach has the following advantages:
 - It minimizes the number of changes, making them more understandable, more manageable, and easier to propagate.
 - It minimizes the number of new problems created during the port.
 - Every change fixes something that is known to be a problem. There are none of the open questions that accompany global replacements.

3.2 Porting Procedures

The following procedures will help in the porting process.

3.2.1 Look for Potential Problems

Use the following methods to locate potential problem areas:

- Use `grep` to search for regular expressions that might indicate problem areas.
- Use `lint -Q`, which checks for common programming techniques that can cause problems when moving from 32-bit systems to 64-bit systems.

3.2.1.1 Use the `grep` Command

Search for the following regular expressions to find constructions that are likely to cause porting problems. (It usually takes less time to find these problems with `grep` rather than with `dbx`.)

- `memcpy`

Byte counts that work on a 32-bit machine might be incorrect on a 64-bit machine. If you pass a variable of type `size_t`, you can avoid this problem.

- `union`

Any union with a `long` is likely to be a problem; a union with an `int` might be a problem.

- `0[xX][0-9a-fA-F]`

Hexadecimal constants, especially when mixed with `long`s or `pointers`, can be problems.

- `%[dDxX]`

Fix the `printf()`/`scanf()` format codes if the parameter is a `long` (for example, `printf("%ld", a_long);`).

- `sizeof(long)`

In Tru64 UNIX this is not the same as `sizeof(int)`.

- `extern`

If an external variable is used, make sure that all declarations of the variable match.

- `long.*32`

Examine lines like the following:

```
typedef long INT32;
```

Examine all `long` declarations; you can convert many of these to `int` to save space. For network addresses, replace an `unsigned long` with an `unsigned int`.

- `<<`, `>>`, and `~`

Add `L` to values acted on by shift operators or complements when those values are `long` constants. This avoids a zero result.

- `&`

Examine the address of `&` to make sure it is not stored in an `int`.

We recommend that you examine all lines of code that have the regular expressions mentioned in this section. However, if the code contains too many occurrences for you to review them all, look at a sampling of these constructs. You might determine that for some constructs, you need to review all the code. You might decide to refine the search characteristics. For example, you could examine all the unions in a particular directory.

3.2.1.2 Use `lint -Q`

Tru64 UNIX provides an option to `lint`, `-Q`, whose sole purpose is to search for programming techniques that might cause problems when moving code from a 32-bit system to a 64-bit system.

You can use a number of suboptions to `-Q` to refine your checking by selectively suppressing checking for specific problem areas. For example, `lint -Qc` suppresses checking for problematic type casts.

For a description of the `-Q` suboptions, see `lint(1)`.

3.2.2 Compile the Code and Save Compiler Messages

When you compile the code with `cc`:

- You can use the `-proto` option to generate function prototypes. Then use `-warnprotos` to check that all functions have prototypes.

The `cc` compiler creates function prototypes automatically when you use the `-proto[is]` option. The `-proto` option extracts prototype declarations for function definitions into a file with a `.H` suffix. The `i` suboption includes identifiers in the prototype, and the `os` suboption generates prototypes for static functions as well.

- Use the `-std1` option to enforce ANSI C standard.
- Use the `-w0` option to display all levels of compiler messages.
- You can deliberately bind addresses to the first 2 gigabytes of address space by specifying the `-tas0` option to `cc` or `ld`. Under this option, the high 33 bits of all pointers are always zero. The `-tas0` option also causes the run-time loader to dynamically relocate shared libraries, such as `libc.so`, into the first 2 gigabytes of address space.

The `-tas0` option might simplify the porting process for applications that assume the size of pointers and variables is always 32 bits.

- The `-xtas0` option can allow applications to use 32-bit pointers. See Section 4.3.5 for more information.

3.2.3 Fix Problems Identified at Compile Time

For each error, the C compiler identifies, as best it can, the line at which the error occurs and the nature of the error.

3.2.4 Fix Alignment and Padding

The Alpha architecture requires that 64-bit quantities (`longs` and pointers) be 64-bit (8-byte) aligned. The Tru64 UNIX operating system corrects unaligned accesses on the fly, but we recommend that you remove unaligned accesses from the code for the following reasons:

- On-the-fly correction of unaligned accesses is substantially slower than normal accesses.
- Unaligned access probably indicates a bug in the program. (In almost all cases, if the code is working correctly, the compiler will arrange things to avoid unaligned accesses.)

See Section 3.3.2 for instructions on finding and correcting the offending line of code. You can find more information on unaligned accesses in the data alignment section of the *Tru64 UNIX Programmer's Guide* and in Appendix A of the *Alpha Architecture Reference Manual*. For information on alignment control options that you can specify on compilation command lines, see `cc(1)`.

3.3 Porting Suggestions

The following suggestions can help you chase down problems in ported code.

3.3.1 Techniques for Finding Problems in Partially Working Code

When you find something wrong and you are done correcting the code and testing the fix, ask yourself, "Could there be similar bugs in similar code elsewhere in the application? What's the best way to find them?" Then use your tool set to find and fix those bugs.

- On tough-to-find problems, a reference platform can be very helpful. On a platform where the application works, run the application in the debugger. Trace through both platforms in parallel and determine where the two diverge.

- Use `dbx` to get a memory dump. For example, a zero is returned where you expected to receive a nonzero value, and you see a pattern in memory that looks like the following:

```
<32-bit value> <32-bit ZERO> <32-bit value> <32-bit ZERO>
```

You might have an array that was created as an array of longs, but then accessed as an array of ints. This makes every other value a zero.

3.3.2 Hints for `dbx`

The following example shows how to find a line of code that caused an unaligned access:

```
Unaligned access pid=3488 <a.out> va=1400001f4 pc=120001178
ra=1200010d8 type=ldq
(dbx) stopi at 0x120001178
(dbx) run [2] stopped at >*[main:7, 0x120001178] ldq r17, 4(r1)
(dbx) where main(0x3ff800f934,0x3ffc00a5840, 0x11ffff788, 0x120001010,
0x1200010d8) ["t.c":7,0x120001178]
```

The argument passing is handled in the following registers:

<code>r0</code>	Return value from a function (nonfloating point)
<code>r16</code>	Function argument 1
<code>r17</code>	Function argument 2
<code>r18</code>	Function argument 3
<code>r19</code>	Function argument 4
<code>r20</code>	Function argument 5
<code>r21</code>	Function argument 6
<code>f16-f21</code>	Floating-point arguments
<code>r31</code>	Hardwired to contain a zero; you cannot move to <code>r31</code>

The first six function arguments are always passed in registers `r16-r21`. If a function has more than six arguments, the extra arguments are passed on the stack.

Most functions, on entry, store the arguments onto the stack. However, simple leaf functions (no calls to other functions) do not store arguments on the stack. Because the `dbx where` command looks on the stack for argument values, the information displayed by the `where` command is incorrect for simple leaf functions.

The following example shows how to dump memory for 20 items, starting at address `0x3000000`. The `dbx assign` command is used to alter the values of variables.

```
(dbx) 0x3000000/20 X longs (64-bit ints)
      x ints (32-bit ints)
      c chars
      i disassembled instructions
(dbx) print *(long *)0x11fffff20
0x632f73756c70322e
(dbx) assign 0x11fffff20 = 17
0x11
```

```
(dbx) print *(long *)0x11ffffff20
0x11
```

Note: 64 bits are changed.

The following example shows how to print a string that starts at address 0x11ffffff20:

```
(dbx) print (char *)0x11ffffff20
```

3.4 Porting Assistant

Compaq Porting Assistant is a set of integrated software tools designed to help port applications to the Tru64 UNIX operating system. It checks for code that is likely to cause porting problems, and it simplifies the process of making needed changes.

See Chapter 5 for more information.

Application Development Environment

The development environment in Tru64 UNIX Version 5.1 is fully compliant with ANSI C and ISO C. It offers the programming features of both BSD and System V UNIX and is compliant with most standards, including POSIX, XPG4, and XPG4-UNIX. Tru64 UNIX features debuggers that support C, Assembler, Fortran (F77 and F90), C++, Ada, and connection to `/proc`. It also supports shared libraries, threads, and versioning, and has a fully optimized C compiler that produces extremely efficient code to exploit the 64-bit address space of the Alpha architecture.

This chapter presents the following topics:

- Software
- C preprocessor
- C compiler
- Linker
- C++
- Lint program checker
- Debugging tools
- System V compatibility
- Other programming tools
- The make command
- Header files
- Shared libraries

4.1 Software

To port or develop applications for Tru64 UNIX, you need the Developers' Toolkit for Tru64 UNIX.

Table 4–1 lists the software subsets for the Tru64 UNIX software development tools and related software. With the exception of `OSFCMPLRSnnn`, which is part of the base Tru64 UNIX operating system, all the other subsets are part of the Developers' Toolkit for Tru64 UNIX.

Table 4–1: Software for Developers

Subset Name	Contents
<code>OSFCMPLRSnnn</code>	Compiler Back End C Language Compiler (This is a mandatory subset.)
<code>OSFSDEnnn</code>	Software Development Tools and Utilities
<code>OSFLIBAnnn</code>	Static Libraries
<code>OSFPGMRnnn</code>	Standard Programmer Commands
<code>OSFINCLUDEnnn</code>	Standard Header Files
<code>OSFSDECDENnn</code>	Software Development Desktop Environment
<code>OSFRTDEVnnn</code>	Realtime Software Development Tools

Table 4–1: Software for Developers (cont.)

Subset Name	Contents
OSFXINCLUDE nnn	X Window and X/Motif Header Files
OSFXDEV nnn	X Window and X/Motif Software Development
OSFXLIBA nnn	X Window and X/Motif Static Libraries
OSFRCS nnn	GNU Revision Control System (RCS)
OSFSCCS nnn	Source Code Control System (SCCS)
OSFLDBBASE nnn	Ladebug Debugger
OSFLDBGUI nnn	Ladebug Debugger Window Interface
OSFXCDAPGMR nnn	CDA Software Development
OSFXCDADEV nnn	CDA for X/Motif Development

In subset names, the sequence nnn indicates the version number of the software; for example, OSFBASE500. The version number depends on the release of Tru64 UNIX you are running. For information about software subsets and about loading software, see the Tru64 UNIX *Installation Guide* and `setld(1)`.

4.2 C Preprocessor

The C preprocessor (`cpp`) on Tru64 UNIX systems is similar to the preprocessor on SCO UNIX systems. Like the SCO UNIX preprocessor, the Tru64 UNIX preprocessor interprets directives, such as `#include` and `#define`. The syntax for specifying directives is the same as the syntax on SCO UNIX systems. The Tru64 UNIX C preprocessor supports additional processor-specific conditions (specified by the `#pragma` directive) and ANSI C preprocessor definitions.

4.3 C Compiler

The Tru64 UNIX operating system provides an ANSI C-compliant compiler. In addition to compiling ANSI C programs, the compiler provides a compilation mode that allows you to compile programs written in Kernighan and Ritchie (K&R) C.

The Tru64 UNIX Version 5.1 C compiler supports 64-bit data types and is NIST-validated for compliance with the ANSI Standard for C. The C front end supports both 64-bit addressing and the interfaces to the System V shared libraries.

This section highlights the features of the Compaq C compiler, and gives some comparisons to the SCO UNIX C compiler.

4.3.1 Features of the Compaq C Compiler

The C compiler offers the following features:

- Compiles the C dialect of the user's choice, including:
 - K&R C (`-std0` mode)
 - Strict ANSI C (`-std1` mode)
 - ANSI C with extensions (`-std` mode)

This mode allows certain extensions to the ANSI standard, such as C++ style comments and casting of the left side of an assignment operator.

- Supports the XPG4-UNIX standard:
 - By default, under the `c89` command
 - With the `-D_XOPEN_SOURCE_EXTENDED` option to `cc`

- Supports floating-point and double-precision operations in the following two modes:
 - IEEE support (including proper handling for exceptional conditions like NaN, INF, and so forth)
 - Fast Math mode (INF, NaN, and so forth, translated to avoid exception handling)
- Supports the following language extensions:
 - C++ style structured exception handling by using `try...except` and termination handling by using `try...finally`. See the chapter on Handling Exception Conditions in the Tru64 UNIX *Programmer's Guide*.
 - User-defined assembly language sequences by using `asm` sequences
 - 32-bit pointers to help reduce the amount of memory used by dynamically allocated pointers, and to facilitate the porting of 64-bit hostile programs
 - Linking programs in 32-bit address space to facilitate the porting of 64-bit hostile programs (the `ld -taso` option)
 - Pragmas for controlling alignment of structures

4.3.2 Support for Interfaces and Definitions

Tru64 UNIX is written using a hierarchy of interfaces and definitions. Using the default interface, `OSF_SOURCE`, applications are able to make use of all the features specified by the OSF Application Environment Specification (AES). If other specific operating system environments are needed, you can use the following macros:

- `POSIX_SOURCE` (for maximum portability of your application)
- `AES_SOURCE`
- `XOPEN_SOURCE`
- `ANSI_C_SOURCE`
- `BSD`

For example, applications needing a POSIX-conforming environment use the `-D_POSIX_SOURCE` compiler switch to compile. Applications needing a strict ANSI-conforming environment use the `-D_ANSI_C_SOURCE` and `-std1` compiler switches to compile.

4.3.3 Compilation Options for Hardware Architectures

The Compaq C compiler supports the `-arch` option for specifying the version of the Alpha architecture for which the instructions are to be generated.

4.3.4 Compilation Options for Checking Code Portability

The Compaq C compiler supports a number of options that can assist you in porting applications:

- `-arch`
Generates instructions that are appropriate for all Alpha processors. Optionally, you can use this to specify optimization for a particular version of the Alpha architecture.
- `-tune`
Selects instruction tuning that is appropriate for all implementations of the Alpha architecture. Optionally, you can use this to specify optimization for a particular version of the Alpha architecture.

- `-check`
Performs compile-time code examination. With this option, the compiler looks for code that exhibits nonportable behavior, represents a possible unintended code sequence, or possibly affects operation of the program because of a subtle change in the ANSI C Standard.
- `-isoc94`
Causes the macro `STDC_VERSION` to be passed to the preprocessor and enables recognition of the digraph forms of various operators.
- `-portable`
Enables the issuance of diagnostics that warn about any nonportable usages encountered. This option is not available when you use the `-oldc` option.
- `-std[n]`
Directs the compiler to produce warnings for language constructs that are not standard in the language. The default is `-std`.
Use `-std1` to enforce strict ANSI C standard.
- `-warnprotos`
Produces warning messages when a function is not declared with a full prototype.
- `-proto[-is]`
Extracts prototype declarations for function definitions and puts them in a `.H` suffixed file. The suboption `i` includes identifiers in the prototype, and the suboption `s` generates prototypes for static functions as well.

For more information on the Compaq C compiler, see `cc(1)`.

4.3.5 Compilation Option for 32-Bit Pointers

The `-xtaso` and `-xtaso_short` compiler options support the use of 32-bit pointers in the 64-bit Tru64 UNIX environment. In addition to simplifying the process of porting applications that make assumptions about the sizes of pointers, the 32-bit pointer data type can help developers reduce the amount of memory used by dynamically allocated pointers.

When you use the `-xtaso` option, all pointer types default to 64-bit pointers. You can declare 32-bit pointers by using `pointer_size` pragmas. Place pragmas where appropriate in your program. Images built with the `-xtaso` option must be linked with the `-taso` option in order to run correctly.

When you use the `-xtaso_short` option, all pointer types default to 32-bit pointers. You can still declare 64-bit pointers by using the `pointer_size` pragmas. Because all system routines continue to use 64-bit pointers, calls to system routines need special handling. You can automate this special handling, with no source changes, by using the appropriate header files. See `protect_headers_setup(8)` for more information.

All `pointer_size` pragmas in a program are ignored unless the program is compiled with either the `-xtaso` or `-xtaso_short` compiler option.

The syntax for the `pointer_size` pragma is as follows:

```
#pragma pointer_size specifier
```

The specifier must be one of the following keywords:

- `long`
All pointers following this pragma are 64 bits in length until an overriding `pointer_size` pragma is encountered.

- `short`
All pointers following this pragma are 32 bits in length until an overriding `pointer_size` pragma is encountered.
- `save`
The current pointer size is saved 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`. The uppermost saved pointer size is restored and deleted from the `save/restore` stack.

For more information about using 32-bit pointers, see Appendix A in the *Tru64 UNIX Programmer's Guide*.

4.3.6 Call Conventions for Passing Arguments

On Tru64 UNIX, the first six function arguments are always passed in registers `r16` to `r21`, with additional arguments passed on the stack. Table 4-2 shows the registers used for passing arguments.

Table 4-2: Call Conventions for Passing Arguments

Register	Description
<code>r0</code>	Return value from a function
<code>r16</code>	Function argument 1
<code>r17</code>	Function argument 2
<code>r18</code>	Function argument 3
<code>r19</code>	Function argument 4
<code>r20</code>	Function argument 5
<code>r21</code>	Function argument 6
<code>\$f16-\$f21</code>	Floating-point registers
<code>r31</code>	Hardwired to contain the value 0

4.4 Linker (ld)

The linker (`ld`) by default loads the program text and data in the high 64-bit virtual address space of the process (between `0xffffffffffffffff` and `0x0000000100000000`).

Currently the virtual address is 43 bits, and the maximum user address is `0x3fffffffffff`. In the kernel, address ranges exist between `0xfffffc0000000000` and `0xfffffe0000000000`, but they are not user accessible. This means no addresses are accessible with a 32-bit address.

If your source code contains any unintended pointer truncations, they will trap into the kernel and cause a run-time error. You can change the default behavior of the linker by using the `-T` or `-D` options to change the text and data segment origin, respectively.

The `ld` Option for Porting Code

The `-taso` option causes the linker to load the executable in the lower 31-bit addressable virtual address range. In addition to setting default addresses for text and data segments, the option causes shared libraries linked outside the 31-bit

address space to be appropriately relocated by the loader. The `-T` and `-D` options, when used in addition to the `-taso` option, override the `-taso` default addresses.

4.5 C++

Compaq C++ Version 6.2 for Tru64 UNIX is based on the ANSI/ISO C++ International Standard, reference designation number ISO/IEC 14882:1998.

To enhance compatibility with other C++ compilers, Compaq C++ supports options that direct the compiler to interpret the source program according to certain rules followed by other C++ compilers. Supported options include the following:

- `-cfront`
Improves compatibility with C++ compilers based on the "cfront" C++ translator from AT&T.
- `-ms`
Improves compatibility with C++ compilers based on Microsoft's Visual C++.

For more information about porting C++ applications, see the manual *Using Compaq C++*.

4.6 The lint Program Checker

The `lint` command examines C programs for bugs, nonportable code, and inefficient code. From a porting perspective, the most useful option may be `-Q`, which turns on checking for programming practices that might cause problems when code is moved from 32-bit systems to 64-bit systems. The `-Q` option examines pointer and data type usage such as the following:

- Pointer alignment
- Pointer and integer data type combinations
- Assignments that cause a truncation of long data types
- Assignments of `long` data types to another type
- Structure and pointer combinations
- Type castings
- Format control strings in `scanf` and `printf` calls

4.7 Debugging Tools

Tru64 UNIX provides four debugging tools:

- `dbx`
A source-level debugger for programs written in C, Fortran, Pascal, assembly language, and machine code.
- `Ladebug`
A symbolic source code debugger for programs compiled by the Compaq C, ACC, Compaq C++, Visual Fortran (Compaq Fortran 90 and Compaq Fortran 77), Compaq Ada, and Compaq COBOL compilers.
Both a graphical interface and command-line interface are available.
- `kdbx`
A crash analysis and kernel debugging tool. It serves as an interactive front end to the `dbx` debugger. The `kdbx` debugger is extensible, customizable, and insensitive to changes of offsets and sizes of fields in structures.
- `ATOM`

A truss-like debugger that supports multithreaded programs.

For more information about debugging tools, see Section 6.9.

4.8 System V Compatibility

Tru64 UNIX provides users, programmers, and administrators with System V functionality.

4.8.1 System V Habitat

Developers who want a System V development environment can use the System V Habitat. The habitat provides source-code compatibility for C-language programs. The habitat is part of the base operating system.

The System V Habitat consists of alternate versions of commands, subroutines, and system calls that support the source code interfaces and run-time behavior for all components of the Base System and Kernel Extension as defined in the System V Interface Definition (SVID). This implementation of the System V Habitat supports all SVID 2 functions and SVID 3 functions. The System V Habitat does not contain alternate versions of default system commands, subroutines, and system calls that already meet the SVID requirement.

For directions on making the System V Habitat your default environment, see the chapter "Using the System V Habitat" in the Tru64 UNIX *Command and Shell User's Guide*. The System V Habitat is described in detail in Appendix B of the Tru64 UNIX *Programmer's Guide*.

4.8.2 Extended System V Compatibility

Prior to Tru64 UNIX Version 5.0, the System V Environment layered software extended the functionality provided by the System V Habitat by supporting a more complete System V Release 4 (SVR4) environment for general users, application programmers, and system administrators. Extended System V compatibility became part of the base operating system in Tru64 UNIX Version 5.0.

4.8.2.1 System V Environment

The System V Environment was an extension to the operating system that contained a separate System V Release 4.0 binary license from UNIX Software Laboratories. A special license and a Product Authorization Key (PAK) were required.

The System V Environment added compliance with the following SVID 3 volumes:

- *Volume 2: Utilities and Administration*
- *Volume 3: Software Development, Terminal Interface, Realtime and Memory Management, Remote Services*

Table 4–3 lists the additional software required for the System V Environment.

Table 4–3: Software for System V Environments

Subset Name	Contents
SVEENV nnn	System V Environment Setup Files Package
SVEADM nnn	System V Environment System Management Package
SVEBCP nnn	System V Environment Base Compatibility Package
SVEDEV nnn	System V Environment API and Development Tools Package

Table 4–3: Software for System V Environments (cont.)

Subset Name	Contents
SVEMAN nnn	System V Environment Reference Pages Package
SVEPRINT nnn	System V Environment Print Package

The System V Environment is described in the Tru64 UNIX *Technical Overview*.

4.8.2.2 Extended System V Compatibility in Tru64 UNIX Version 5.0

Starting with Tru64 UNIX Version 5.0, the features of the System V Environment are integrated into the base operating system. Tru64 UNIX provides 80 percent of the SVID standard, as verified by System V Verification Suite 3 (SVVS 3) and System V Verification Suite 4 (SVVS 4). As a result, Tru64 UNIX contains a substantial number of SVR4 features and delivers the highest composite SVR4 conformance of any implementation.

SVR4 functionality will be expanded in future releases. For details, see the Software Product Description for your particular version of Tru64 UNIX.

4.9 Other Programming Tools

Tru64 UNIX provides other programming tools that are also available on SCO UNIX. Each tool has been modified to support the ANSI C language dialect, shared libraries, and 64-bit data types. Otherwise, its use is the same as its SCO UNIX equivalent, when one exists.

Table 4–4 gives a brief description of each Tru64 UNIX tool. For more information about the tools, see the respective reference pages.

Table 4–4: Tru64 UNIX Programming Tools

Tool	Function
ar	Creates and maintains archive libraries. (You cannot use the ar command to create shared libraries.)
atom:	Profiling tools.
hiprof	Produces flat and hierarchical profile of execution times.
pixie	Produces profile by procedure, source line, or instruction.
third	Performs memory access checks and detects leaks.
cflow	Analyzes C application files (as well as yacc, lex, and assembler files) and builds a graph that charts the external references made in the application.
ctags	Creates a tags file that you can use with the ex editor. The tags file specifies the location of functions and typedef declarations in the specified set of C application files.
cxref	Analyzes a set of C application files and builds a cross-reference table. The table lists all symbols used in the application.
dis	Disassembles object files into machine instructions.
gprof	Produces an execution profile.
ldd	Lists the dynamic dependencies of an executable file or shared object.
lex	Generates a C language source file that matches patterns for simple lexical analysis of an input stream.
nm	Displays symbol table information for object files, archive files and shared libraries.

Table 4–4: Tru64 UNIX Programming Tools (cont.)

Tool	Function
file	Reads one or more files as input, performs a series of tests on the files, and determines their types.
odump	Displays information about an object file, archive file, or executable file. You can use <code>odump</code> options to display an object file's header, defined symbols, or program regions.
prof	Analyzes profile data.
pixstats	Analyzes the output from <code>pixie</code> ; supported only with archive libraries and cannot be used with shared libraries.
rcs	Revision Control System (RCS).
sccs	Source Code Control System (SCCS).
size	Displays the number of bytes required by each section of an object file, as well as the total number of bytes required by an object file.
stdump	Displays detailed symbol table information for an application or object.
strip	Strips symbolic debugger information from an executable file.
trace	Monitors variable accesses.
uprofile kprofile	Profiles a program (<code>uprofile</code>) or the kernel (<code>kprofile</code>) by using built-in performance counters on the Alpha processor.
yacc	Converts a context-free grammar specification into a set of tables that a simple parsing program can use.

4.10 The make Command

Tru64 UNIX provides three versions of the `make` command:

- `/bin/make`
The default. For a detailed description, see `make(1)`.
- `/usr/bin/posix/make`
The XPG4-UNIX version. For a detailed description, see `make(1p)`.
- `/usr/opt/ultrix/usr/bin/make`
The System V version with some Berkeley compatibility added. For a detailed description, see `make(1u)`.

In addition to the appropriate reference page, you can find information about the `make` command in the Tru64 UNIX manual *Programming Support Tools*.

Within the Tru64 UNIX Workstation Development Environment, the `imake` and `makedepend` tools are available as a `make` preprocessor utility and a `make` dependency tool.

The description that follows applies to the default version of the command, `/bin/make`.

4.10.1 The makefile Search Path

If the `make -f` option is not used, then `make` looks for makefiles in the locations indicated in Table 4–5. Tru64 UNIX assumes RCS, not SCCS, as the default for file archiving.

Table 4–5: The makefile Search Path

UnixWare make	Tru64 UNIX make
makefile	makefile
Makefile	Makefile
SCCS/s.makefile	makefile,v
SCCS/s.Makefile	Makefile,v
	RCS/makefile,v
	If the RCS file is present, Tru64 UNIX attempts to perform a checkout.
	RCS/Makefile,v
	If the RCS file is present, Tru64 UNIX attempts to perform a checkout.

4.10.2 Options for the make Command

Table 4–6 shows differences between UnixWare make options and those in Tru64 UNIX.

Table 4–6: Options to the make Command

Option	UnixWare Function	Tru64 UNIX Equivalent
-B	Arranges the output of parallel make in appropriate blocks for readability.	No equivalent
-p	Updates multiple target in parallel.	No equivalent
-u	Unconditionally makes the target, regardless of timestamps.	The -u option in Tru64 UNIX performs a different function from that option in UnixWare.
-w	Suppresses warning messages.	No equivalent

4.10.3 Implementation of make Rules

Tru64 UNIX has the default rule implementation in the make binary, and uses a Makeconf file to supersede the default rules with any compilation-specific changes. Tru64 UNIX automatically searches for a Makeconf file in the current directory.

4.10.4 Make Directives

What SCO UNIX documentation refers to as make directives, Tru64 UNIX documentation calls pseudotargets. Table 4–7 summarizes the behavior of make directives in Tru64 UNIX.

Table 4–7: Special-Function Targets

Make Directive	Behavior in Tru64 UNIX
.SUFFIXES	Work similarly to those in SCO UNIX, but the suffixes list is different. The default suffixes list in Tru64 UNIX follows: .out .o .s .c .a .F .f .e .r .y .yr .ye .l .p .sh .csh .h .f~ .f90~
.DEFAULT	Equivalent on both systems.
.IGNORE	Equivalent on both systems.
.PRECIOUS	Equivalent on both systems.

Table 4–7: Special-Function Targets (cont.)

Make Directive	Behavior in Tru64 UNIX
.SILENT	Equivalent on both systems.
.MUTEX	No equivalent.

4.10.5 Built-in Macros

The Tru64 UNIX `make` command recognizes the built-in macros `$$`, `$$@`, `$$?`, `$$<`, and `$$*`. It does not recognize the `$$%` built-in macro. That macro is available in the XPG4-UNIX version of `make`, `/usr/bin/posix/make`.

4.11 Header Files

The locations and contents of header files differ among the various UNIX operating systems. These differences can appear in a number of ways. For example, the interface to a service might be slightly different, structure definitions might be located in different header files, values might have changed to reflect the 64-bit Alpha architecture, or nearly identical structures or constants might have different names.

The Tru64 UNIX header files are accessible in the `/usr/include` directory. Some subdirectories in `/usr/include` are links to subdirectories in `/usr/sys/include`.

Table 4–8 describes the Tru64 UNIX directories that contain standard header files.

Table 4–8: Tru64 UNIX Standard Header File Directories

Directory	Description
<code>dec</code>	Header files specific to Tru64 UNIX
<code>DPS</code>	Display PostScript System C header files
<code>DXm</code>	Header files with Tru64 UNIX extensions to Motif C
<code>lvm</code>	C header files for Logical Volume Manager (LVM)
<code>mach</code>	Mach-specific C include files
<code>Mrm</code>	Motif resource manager C header files
<code>net</code>	Miscellaneous network C header files
<code>netimp</code>	C header files for IMP protocols
<code>netinet</code>	C header files for Internet standard protocols
<code>netns</code>	C header files for XNS standard protocols
<code>nfs</code>	C header files for Network File System (NFS)
<code>protocols</code>	C header files for Berkeley service protocols
<code>rpc</code>	C header files for remote procedure calls
<code>servers</code>	C header files for servers
<code>sys</code>	System C header files (kernel data structures)
<code>tli</code>	C header files for Transport Layer Interface (TLI)
<code>ufs</code>	C header files for UNIX File System (UFS)

Table 4–8: Tru64 UNIX Standard Header File Directories (cont.)

Directory	Description
uil	Header files for the User Interface Language (UIL) Compiler
X11	X Toolkit header files
Xm	Motif C header files

The Compaq C compiler can help you port your application by finding inconsistencies in the application's use of a symbol, a function, or a declaration in a header file. The compiler issues error messages for the following conditions:

- **Header file not found**

```
cfe: Error: file.c: 1: Cannot open file cursesX.h for
#include
```

- **Undefined symbol (a symbol that is not defined before its use)**

This message helps you to find references to header file symbols that have moved or are no longer available.

```
cfe: Error: file.c, line 8: 'ENOSYSTEM' undefined,
reoccurrences will not be reported
```

- **Multiple-defined symbol (a local definition that conflicts with a header file definition)**

```
cfe: Warning: file.c:4: Tried to redefine the macro
EDEADLK, this macro keeps the old definition in
std/stdl mode, otherwise the macro is redefined
```

- **Redeclared function (a local function declaration that conflicts with a header file declaration)**

```
cfe: Error: t.c, line 7: redeclaration of 'openlog';
previous declaration at line 120 in file
'/usr/include/syslog.h'
int openlog(char*, int);
----^
```

- **Mismatched function use and prototype (failure of a function usage to provide the number of arguments declared by the prototype declaration)**

```
cfe: Error: file.c, line 12: Number of arguments
doesn't agree with number in declaration
```

- **Incompatible function arguments (an attempt to provide incompatible arguments to a function)**

```
cfe: Warning: file.c, line 12: Incompatible pointer
type assignment
```

Because function declarations or prototypes are not required by the C language before a function call, the compiler cannot detect misuse of functions that did not have a preceding prototype declared. To find differences in these cases, first determine which header files your application depends on, generate a list of the function declarations these header files contain, and then use this list of functions to generate a cross-reference for the needed header files on a Tru64 UNIX system. Then you can cross-check the actual declarations for changes in the function interfaces and modify your source code where necessary. You can use shell scripts to search for the appropriate definitions in the list of header files.

4.12 Shared Libraries Provided by Tru64 UNIX

Table 4–9 describes the shared libraries that Tru64 UNIX provides in the `/usr/shlib` directory.

Table 4–9: Shared Libraries in /usr/shlib

/usr/shlib Library	Description
libDXm.so	Motif Extensions library.
libDXterm.so	DECterm widget library, used by dxterm.
libDtHelp.so	CDE online help routines.
libDtMail.so	Shared library support for the dtmail CDE mail utility.
libDtSvc.so	CDE service routines for desktop management.
libDtTerm.so	Shared library support for the CDE ddterm terminal emulator utility.
libDtWidget.so	Shared library of CDE widgets to supplement the Motif widget.
libICE.so	Inter-Client Exchange library, which enables the building of protocols.
libMrm.so	Motif Resource Manager library.
libSM.so	The X Session Management Protocol (XSMP), which provides a uniform mechanism for users to save and restore their sessions using the services of a network-based session manager. It is built on ICE and is the C interface to the protocol.
libUil.so	The callable Motif UIL (User Interface Language) compiler used by applications that want to compile UIL at run time.
libX11.so	Xlib library.
libXETrap.so	X Extension library.
libXaw.so	X Athena Widgets run-time library.
libXext.so	X Extension client-side library.
libXi.so	X Input Extension client-side library.
libXIE.so	X Imaging Extension client-side run-time library (V5).
libXie.so	X Imaging Extension client-side run-time library (V3).
libXm.so	Motif Widgets library.
libXmu.so	X Miscellaneous utilities run-time library.
libXt.so	X Intrinsics library.
libXtst.so	A library of routines for X clients to make use of the XTEST Extension.
libXv.so	X video Extension client-side run-time library.
libaio.so	POSIX realtime asynchronous I/O functions.
libaio_raw.so	POSIX realtime asynchronous I/O functions (raw disk and tape only).
libaud.so	C2 security auditing library.
libbkr.so	Motif Help System library.
libc.so	C library.
libc_r.so	Threadsafe libc (link to libc.so).
libcda.so	CDA run-time library.
libcdrom.so	Rock Ridge Extensions to CDFS library.
libchf.so	CDA/Imaging signal-handling routines.
libcma.lib.so	CMA threads library.

Table 4–9: Shared Libraries in /usr/shlib (cont.)

/usr/shlib Library	Description
libcsa.so	Shared library portion of the CDE dtcm calendar manager utility.
libcurses.so	Curses screen control library.
libcxx.so	C++ run-time support library.
libdb.so	Database routines.
libdnet_stub.so	DECnet library.
libdps.so	Adobe Display PostScript client-side run-time libraries.
libdpstk.so	Adobe Display PostScript toolkit.
libdvr.so	CDA run-time viewer library.
libdvs.so	CDA run-time layout library.
libesnmp.so	Extensible SNMP library.
libexc.so	Library that provides support for exception handling.
libiconv.so	Internationalization codeset conversion routines.
libids.so	Image-display services library.
libids_nox.so	Image-display services not dependent on X.
libimg.so	Image-processing routines.
libips.so	Image-processing routines.
libm.so	Compaq Portable Mathematics Library (CPML).
libmach.so	Mach library.
libmxr.so	Library used by mxr, the ULTRIX binary interpreter for OSF/1.
libndb.so	Database routines.
libots.so	Compiler run-time support.
libpacl.so	POSIX Access Control List library.
libproplist.so	VFS Extended File Attributes library.
libpset.so	Processor set routines.
libpsres.so	Adobe Display PostScript resource utilities.
libpthread.so	Application Programming Interface for Tru64 UNIX threads.
libpthread.so	POSIX Threads library.
libsecurity.so	C2 security library.
libsm_x.so	Systems Management Graphical support library; no user-level interfaces available.
libtcl.so	Base Tool Command Language (TCL) support library.
libtclx.so	Extended TCL support library.
libtk.so	Graphical TCL (TK) Extensions library.
libtkx.so	Graphical Extended TCL support library.
libtli.so	XTI library.
libtt.so	SunSoft Tooltalk routines.
libvxvm.so	LSM utility library.
libmsfs.so	AdvFS system call interface library.

Table 4–9: Shared Libraries in /usr/shlib (cont.)

/usr/shlib Library	Description
libfilsys.so	File system utility library.
libxnet.so	X/Open networking interface library.
libxti.so	XTI library.

Table 4–10 describes the shared libraries that Tru64 UNIX provides in the /usr/shlib/X11 directory.

Table 4–10: Shared Libraries in /usr/shlib/X11

/usr/shlib/X11 Library	Description
libXau.so	X Authorization library
libXdmDecGreet.so	Motif loadable greeter library
libXdmGreet.so	Athena-style loadable greeter library
libXdmcp.so	X Display Manager control program library
lib_adobe_dps.so	Adobe Display PostScript Extension library
lib_dec_cirrus.so	Device support for the Cirrus VGA graphics card
lib_dec_ffb.so	Support for the sfb+ graphics accelerator for 2D and 3D drawing operations
lib_dec_sfb.so	Device support for the smart frame buffer (HX)
lib_dec_smt.so	Shared memory transport library
lib_dec_tx.so	Device support for the TX graphic adapter
lib_dec_ws.so	Low-layer operating system interface for the X Server
lib_dec_xi_pcm.so	Dynamically loadable X Input Extension library that supports the dial and box
lib_dec_xi_serial_mouse.so	Support library for the serial mouse
lib_dec_xv_tx.so	X Video Extension support for the TX graphic option
libcfb.so	Color frame buffer library
libcfb16.so	16-bit visual support for the color frame buffer
libcfb32.so	32-bit visual support for the color frame buffer
libdbe.so	DOUBLE-BUFFER Extension library
libdix.so	Device-independent portion of the X Server
libdixie.so	With libmixie.so, supports the X Image Extensions (XIE) library
libextMITMisc.so	MIT-SUNDRY-NONSTANDARD Extension library
libextMultibuf.so	MultiBuffering Extension library
libextScrnsvr.so	MIT-SCREEN-SAVER Extension library
libextSync.so	SYNC Extension library
libextXCMisc.so	XC-MISC Extension library
libextbigreq.so	BIG-REQUESTS Extension library
libextkme.so	Keyboard-Management-Extension library
libextshape.so	SHAPE Extension library
libextshm.so	MIT-SHM Extension library
libextxtest.so	XTEST Extension library

Table 4–10: Shared Libraries in /usr/shlib/X11 (cont.)

/usr/shlib/X11 Library	Description
libexttrap.so	DEC-XTRAP Extension library
libfont.so	Font access library
libfr_Speedo.so	Loadable font renderer library
libfr_Type1.so	Loadable font renderer library
libfr_fs.so	Loadable X Server font renderer for using a font server
libmfb.so	Monochrome frame buffer support
libmi.so	Machine-independent portion of the X Server
libmixie.so	With libdixie.so, supports the X Image Extensions (XIE) Extension library
libos.so	Operating system-dependent portion of the X Server
libxinput.so	X Input Extension server-side library
libxkb.so	XKEYBOARD Extension library

4.12.1 Tru64 UNIX Archive Libraries

In addition to shared libraries, the Tru64 UNIX system provides archive libraries. When you link your application with them, the image for library routines you call is included in your application image. You can link Tru64 UNIX applications to either shared libraries or archive libraries.

Use of shared libraries, rather than archive libraries, is encouraged. Normally, you can use shared libraries in any application and create any library as a shared library. In most cases, the effect of using shared libraries instead of archive libraries is transparent.

4.12.2 Using Shared Libraries

The use of shared libraries is transparent to the user.

To turn off the use of shared libraries, use the `-non_shared` option to the `cc` command. The following example creates a nonshared executable on Tru64 UNIX by using the `/usr/lib/libc.a` static archive library:

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

Porting Assistant

This chapter describes Porting Assistant, a tool designed specifically to help you port source code to Tru64 UNIX from other UNIX and non-UNIX platforms. It runs on Tru64 UNIX Versions 4.0 and higher.

5.1 What Porting Assistant Does

Porting Assistant can check C, C++, and Fortran code to identify segments that might not compile or run properly on a Tru64 UNIX system. It generates diagnostic messages indicating the potential problems. The specific code in question is displayed in an editor window along with the corresponding diagnostic message. HyperHelp is available to explain why changes need to be made. Tools are provided to simplify the process of making corrections. Porting Assistant also makes it easier to link code by helping to resolve undefined symbols.

5.1.1 Code Checking

Porting Assistant checks for the following potential porting problems:

- Conditional code (`#ifdef`) that might need a Tru64 UNIX branch
If the application already uses conditional code that applies to one or more platforms, Porting Assistant can locate this conditional code. You can then decide how to handle that code for Tru64 UNIX.
- Includes (`#include`) of files that do not exist in the specified directories
Porting Assistant identifies include files that are not found in the specified location. If possible, it reports locations where the files do exist.
- Calls to library functions that do not match Tru64 UNIX definitions
Porting Assistant checks for functions that do not have function definitions in Tru64 UNIX. For many of these functions, mappings to equivalent Tru64 UNIX functions are provided.
Porting Assistant also flags functions that have calls that are inconsistent with their definition. This can occur if a function has a different signature on Tru64 UNIX than on another platform.
- Calls to library functions with different semantics on Tru64 UNIX
Porting Assistant provides a list of functions that have different semantics on Tru64 UNIX than on other platforms. Categories of function differences include platform differences and 32-bit and 64-bit differences. For each such function, Porting Assistant provides access to the function's reference page and to a special reference page that describes how this function is different on Tru64 UNIX. Porting Assistant searches your code to find calls to these functions.
- Code with 32-bit dependencies
Porting Assistant locates code that assumes it is running on a 32-bit architecture. This includes such operations as assigning or casting pointers to integers, assigning `longs` to other types, and sign-extension problems.
- Commands in `makefiles` that might not run correctly on Tru64 UNIX
Porting Assistant can check a `makefile` to ensure that all specified actions exist on the user's path. If Porting Assistant cannot find an action, it suggests

locations of a program with the same name. Porting Assistant checks the arguments and options to common build commands (`cc`, `ld`, `ar`, and `rm`). It reports when the Tru64 UNIX option is different from that on the source platform and gives the equivalent Tru64 UNIX option.

5.1.2 Diagnostic Messages

A Porting Assistant diagnostic message reports the name of the file and the line number where the potential problem was found, the severity of the problem, and brief text describing the nature of the problem.

Examples of diagnostic text include:

- Assignment of `longs` to other types
- Illegal combinations of `pointer` and data types
- Shift operator yields zero
- Truncation of `longs` in assignments
- Function calls with no prototype
- Function name is incorrectly called with n arguments

If you find the meaning of a diagnostic message unclear, you can use HyperHelp to obtain an explanation of the message and a suggested course of action.

5.1.3 HyperHelp on Porting

Porting Assistant includes online HyperHelp that explains how to use Porting Assistant and its checks. You can navigate through HyperHelp by task or by the index, or you can perform a search for the information.

The HyperHelp also includes porting tips that cover topics such as byte ordering, data alignment, and shared libraries. The extensive HyperHelp for the diagnostic messages is particularly useful because it is directly linked to specific messages with a description of the problem and suggestions on how to fix it. In addition to HyperHelp, you can access reference pages from Porting Assistant.

5.1.4 Code Correction

Porting Assistant displays the code in question in the editor of your choice: `vi`, `emacs`, or a bundled Motif editor. The associated diagnostic message is also displayed. An editor option lets you compile the code in the editor buffer to test changes quickly.

To speed the task of changing function names, variable names, or type names in source files, Porting Assistant lets you change a specified string to a different string in a set of files. Porting Assistant offers the option of performing substitution with or without confirmation. If you want to confirm each substitution, Porting Assistant brings each file into an editor, with the cursor positioned on each occurrence of the specified string. You can then choose to change that occurrence of the string.

5.1.5 Linker Assistance

Porting Assistant helps resolve undefined symbols reported by the linker. It includes a facility to look up a function to see what library defines it. It reports the library name and the linker options that are needed to link the library with the application.

5.2 How to Use Porting Assistant

This section is intended as an introduction to using Porting Assistant. After you have Porting Assistant running, you will likely find it intuitive to use. If you have questions, you can access HyperHelp by clicking on the Help button that is available on all Porting Assistant windows.

5.2.1 Getting Started

Porting Assistant ships with the Developers' Toolkit for Tru64 UNIX Version 4.0 and later. The subset name for Porting Assistant Version 3.0 is `PRTBASE300`.

You can download the latest version of Porting Assistant from the following location:

<http://www.tru64unix.compaq.com/portingassistant/>

To function, Porting Assistant requires that the Developers' Toolkit for Tru64 UNIX be installed on the system.

After you install Porting Assistant, you can start it from the command line as follows:

```
# port &
```

The Porting Assistant Control Panel appears. From the File pulldown menu, you can choose to work on a new project or to open an existing project.

If you choose New, you are presented with the Project Manager window, where you specify the following information:

- Project name
- Working directory for the project
- Directory for the executable target
- Directory for data files
- Directory for the current source to be ported
- Directory for the ported source

After you supply the necessary information and dismiss the Project Manager window, you return to the Porting Assistant Control Panel. If the name for your project is not already highlighted, select it by placing the cursor on the name and clicking MB1.

Go to the Tools pulldown menu and choose Porting Assistant. The Application Information dialog box appears. Provide the requested information. Indicate whether you are using an existing `makefile` or creating a new one. If you are creating a new `makefile`, the Application Information dialog box assists you in the process.

When you finish with the Application Information dialog box, the Porting Assistant window appears. This window is central to using Porting Assistant. From it, you select and run the various checks that help you to port your source code.

5.2.2 Performing Checks

Regardless of which check you want, you perform the following actions:

1. Run the check.
2. Step through the source code corresponding to each diagnostic message.
3. Make any needed corrections.

5.2.2.1 Run the Check

Select the type of check you want and run it:

- Ifdef Code
- Include Files
- 32-Bit Dependencies
- Function Calls
- Parallel Directives
- Platform-Specific Assumptions
- Makefile

Porting Assistant uses a tool called the builder to run the check.

The main area of the builder is a transcript of the check. You can see the builder progressing through the files being ported. Any diagnostic messages are reported first in the builder.

5.2.2.2 Step Through the Source Code

When the check finishes, you walk through the list of diagnostic messages by using the First, Next, Previous, and Last buttons.

When you click on the First button, the builder launches the editor to display the source code corresponding to the first diagnostic message.

Porting Assistant offers you a choice of a Motif-based editor, `emacs`, or `vi`. The left margin of the editor is the annotations area. A "B" in this area indicates a line of source code corresponding to a diagnostic message.

Markings that can appear in the annotations area include:

- An exclamation point on a green background, which indicates that the diagnostic message is informational rather than an error
- An exclamation point on a yellow background, which indicates that the diagnostic message refers to an error
- A square or hexagon, indicating a break point

The editor displays these annotations only if it is already running when the user starts the check.

The editor provides a Find Annotation function to let you move quickly through code to annotations of interest.

You can use the Editor Filter function to filter out unwanted diagnostic messages, as in the following examples:

- If after assessing the source code flagged with a diagnostic message, you determine that you do not need to view future occurrences of that same message, you can use the Filter Instance function to filter the message in that context from future builds.
- If you determine that you do not need to view occurrences of that message as applied anywhere in the current file, you can use the Filter File function.
- If you determine that you do not need to view future occurrences of that message in any of your files, you can use the Filter Everywhere function.

5.2.2.3 Make the Correction

The Motif-based FUSE editor is a fully functional editor. In addition to the Find feature, available on the editor window Buffer pulldown menu, Porting Assistant offers a Search option on the Tools pulldown menu on the editor window and on the other major Porting Assistant windows. The Search window allows multifile search and replace.

5.2.2.4 Repeat the Process

After you make corrections for one of the code checks, return to the Porting Assistant window. Select the next code check appropriate to your porting job, and repeat the process of making corrections.

5.3 Possible Limitations

Some functions in Porting Assistant depend on information in Porting Assistant databases. Follow-on releases of a vendor's product might make such information out of date: for example, the reference pages that show differences between a routine as implemented in Tru64 UNIX and another vendor's UNIX.

The following functions in Porting Assistant might be rendered outdated or incomplete because of changes in a vendor's UNIX after Porting Assistant shipped:

- Find Function on Tru64 UNIX (from the editor Utilities pulldown menu)
The Function Finder locates functions on Tru64 UNIX that are equivalent to functions on the platform from which the code is being ported. The introduction of new functions by a vendor might result in the Function Finder failing to find an equivalent function in Tru64 UNIX.
- Code Check for Platform-Specific Assumptions
The list of possible functions to check for might be incomplete if a vendor introduces new functions that involve assumptions about the hardware platform they run on.

Despite the possible limitations previously described, Porting Assistant provides a number of features that will speed the porting process. Even assuming the worst case – a port from an implementation of UNIX about which Porting Assistant databases have no specific information – Porting Assistant retains most of its functionality.

Porting Threaded Applications

This chapter is intended to help you begin the process of porting threaded applications to Tru64 UNIX. For detailed information about pthreads in Tru64 UNIX, see the reference pages for specific pthread functions and the *Guide to the POSIX Threads Library* (formerly the *Guide to DECthreads*).

A number of good books on pthreads programming are available. Among them are *Programming with POSIX Threads*, by David R. Butenhof, Addison-Wesley Publishing Co.; and *Pthreads Programming* by Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell, O'Reilly & Associates, Inc.

The POSIX Threads Library pthread interface is an implementation of the POSIX standard 1003.1c-1995 (part of 1003.1-1996). The POSIX Threads Library adds extensions specified in The Open Group's Single UNIX Specification, Version 2 (SUSV2), also known as XSH5, part of the UNIX 98 brand. Although the POSIX Threads Library provides nearly all of the threads-related extensions specified by SUSV2, technically it is not SUSV2 compliant.

Until recently, SCO offered no supported threads API for UnixWare. With UnixWare Version 2, SCO introduced a threads package based on the System V Release 4.2 MP "UI threads" interface. This package originated several years ago when Sun Microsystems designed it in cooperation with UNIX International. Sun has since moved to POSIX threads, although it still supports Solaris threads to provide backward compatibility for older applications.

Two thread packages are available for OpenServer 5:

- POSIX 1003.4a Draft 4, also known as DECthreads. This package is a separately priced layered product.
- An unsupported shareware pthreads. Based on the proposed POSIX 1003.4a Draft 8, this unsupported package is available free on the skunkworks CD-ROM.

Both of these drafts for a threads API were superseded several years ago by the POSIX Standard 1003.1c, which the POSIX Threads Library follows. To provide backward compatibility, the API for POSIX 1003.4a Draft 4 threads is available in Tru64 UNIX in the `libpthreads.so` shared library. However, it is recommended that you migrate to POSIX Standard 1003.1c.

This chapter presents the following topics:

- Porting applications that use POSIX threads
- Porting applications that use DCE threads
- Porting applications that use UnixWare threads
- Semaphores
- Programming notes
- Files
- Linking
- Compiling
- Debugging

6.1 Porting Applications That Use POSIX Threads

POSIX Draft 8 1003.4a is very similar to the final POSIX Standard 1003.1c. There might be some differences in the return values of certain thread library calls. However, porting a threaded application that uses the POSIX Draft 8 1003.4a threads API to use POSIX Threads Library should be a straightforward process as far as the threading aspect is concerned.

6.2 Porting Applications That Use DCEthreads

This section presents the differences between DCEthreads and POSIX threads.

Table 6–1 maps the DCEthreads API to the POSIX Threads Library routines. Although the functions of these routines are the same, there might be differences in syntax and return values. See the reference pages for specifics.

Table 6–1: Mapping DCEthreads to POSIX Threads Library Routines

Draft 1003.4a Routines	Corresponding POSIX Threads Library Routines
<code>pthread_attr_create()</code>	<code>pthread_attr_init()</code>
<code>pthread_attr_delete()</code>	<code>pthread_attr_destroy()</code>
<code>pthread_attr_getprio()</code>	<code>pthread_attr_getschedparam()</code>
<code>pthread_attr_setprio()</code>	<code>pthread_attr_setschedparam()</code>
<code>pthread_attr_getsched()</code>	<code>pthread_attr_getschedpolicy()</code>
<code>pthread_attr_setsched()</code>	<code>pthread_attr_setschedpolicy()</code>
<code>pthread_condattr_create()</code>	<code>pthread_condattr_init()</code>
<code>pthread_condattr_delete()</code>	<code>pthread_condattr_destroy()</code>
<code>pthread_keycreate()</code>	<code>pthread_key_create()</code>
<code>pthread_mutexattr_create()</code>	<code>pthread_mutexattr_init()</code>
<code>pthread_mutexattr_delete()</code>	<code>pthread_mutexattr_destroy()</code>
<code>pthread_mutexattr_getkind_np()</code>	<code>pthread_mutexattr_gettype_np()</code>
<code>pthread_mutexattr_setkind_np()</code>	<code>pthread_mutexattr_settype_np()</code>
<code>pthread_setasynccancel()</code>	<code>pthread_setcanceltype()</code>
<code>pthread_setcancel()</code>	<code>pthread_setcancelstate()</code>
<code>pthread_getprio()</code>	<code>pthread_getschedparam()</code>
<code>pthread_setprio()</code>	<code>pthread_setschedparam()</code>
<code>pthread_getscheduler()</code>	<code>pthread_getschedparam()</code>
<code>pthread_setscheduler()</code>	<code>pthread_setschedparam()</code>
<code>pthread_yield()</code>	<code>sched_yield()</code>

6.2.1 Routines with Syntax Changes

Names for the routines in this section are unchanged between the POSIX 1003.4a draft 4 and 1003.1c, but the syntax has changed. Refer to the appropriate reference pages for details.

6.2.1.1 `pthread_attr_getinheritsched()`

POSIX 1003.4a Draft 4

```
int pthread_attr_getinheritsched(
    pthread_attr_t attr);
```

POSIX Standard 1003.1c

```
int pthread_attr_getinheritsched(  
    const pthread_attr_t *attr,  
    int *inheritsched);
```

6.2.1.2 pthread_attr_getstacksize()

POSIX Draft 1003.4a Draft 4

```
long pthread_attr_getstacksize(  
    pthread_attr_t attr);
```

POSIX Standard 1003.1c

```
int pthread_attr_getstacksize(  
    const pthread_attr_t *attr,  
    size_t *stacksize);
```

6.2.1.3 pthread_attr_setstacksize()

POSIX 1003.4a Draft 4

```
int pthread_attr_setstacksize(  
    pthread_attr_t *attr,  
    long stacksize);
```

POSIX Standard 1003.1c

```
int pthread_attr_setstacksize(  
    pthread_attr_t *attr,  
    size_t stacksize);
```

6.2.1.4 pthread_cleanup_pop()

POSIX 1003.4a Draft 4

```
void pthread_cleanup_pop(  
    int execute);
```

POSIX Standard 1003.1c

```
int pthread_cleanup_pop(  
    int execute);
```

6.2.1.5 pthread_cleanup_push()

POSIX 1003.4a Draft 4

```
void pthread_cleanup_push(  
    void routine,  
    pthread_addr_t arg);
```

POSIX Standard 1003.1c

```
int pthread_cleanup_push(  
    void (*routine)(void *),  
    void *arg);
```

6.2.1.6 pthread_cond_init()

POSIX 1003.4a Draft 4

```
int pthread_cond_init(  
    pthread_cond_t *cond,  
    pthread_condattr_t attr);
```

POSIX Standard 1003.1c

```
int pthread_cond_init(
    pthread_cond_t      *cond,
    const pthread_condattr_t *attr);
```

6.2.1.7 pthread_create()

POSIX 1003.4a Draft 4

```
int pthread_create(
    pthread_t *thread,
    pthread_attr_t attr,
    pthread_startroutine_t start_routine,
    pthread_addr_t arg);
```

POSIX Standard 1003.1c

```
int pthread_create(
    pthread_t      *thread,
    const pthread_attr_t *attr,
    void *         (*start_routine)(void *),
    void          *arg);
```

6.2.1.8 pthread_detach()

POSIX 1003.4a Draft 4

```
int pthread_detach(
    pthread_t      *thread);
```

POSIX Standard 1003.1c

```
int pthread_detach(
    pthread_t      thread);
```

6.2.1.9 pthread_equal()

POSIX 1003.4a Draft 4

```
boolean32 pthread_equal(
    pthread_t      *thread1,
    pthread_t      *thread2);
```

POSIX Standard 1003.1c

```
int pthread_equal(
    pthread_t      t1,
    pthread_t      t2);
```

6.2.1.10 pthread_exit()

POSIX 1003.4a Draft 4

```
void pthread_exit(
    pthread_addr_t      status);
```

POSIX Standard 1003.1c

```
void pthread_exit(
    void          *value_ptr);
```

6.2.1.11 pthread_getspecific()

PSIX 1003.4a Draft 4

```
int pthread_getspecific(
    pthread_key_t      key,
```

```
pthread_addr_t *value);
```

POSIX Standard 1003.1c

```
void pthread_getspecific(  
pthread_key_t key);
```

6.2.1.12 pthread_join()

POSIX 1003.4a Draft 4

```
int pthread_join(  
pthread_t thread,  
pthread_addr_t *status);
```

POSIX Standard 1003.1c

```
int pthread_join(  
pthread_t thread,  
void **value_ptr);
```

6.2.1.13 pthread__lock_global_np()

POSIX 1003.4a Draft 4

```
void pthread_lock_global_np();
```

POSIX Standard 1003.1c

```
int pthread_lock_global_np(void);
```

6.2.1.14 pthread_mutex_init()

POSIX 1003.4a Draft 4

```
int pthread_mutex_init(  
pthread_mutex_t *mutex,  
pthread_mutexattr_t attr);
```

POSIX Standard 1003.1c

```
int pthread_mutex_init(  
pthread_mutex_t *mutex,  
const pthread_mutexattr_t *attr);
```

6.2.1.15 pthread_once()

POSIX 1003.4a Draft 4

```
int pthread_once(  
pthread_once_t *once_block,  
pthread_initroutine_t init_routine);
```

POSIX Standard 1003.1c

```
int pthread_once(  
pthread_once_t *once_control,  
void (*init_routine)(void));
```

6.2.1.16 pthread_setspecific()

POSIX 1003.4a Draft 4

```
int pthread_setspecific(  
pthread_key_t key,  
pthread_addr_t value);
```

POSIX Standard 1003.1c

```
int pthread_setspecific(
    pthread_key_t key,
    const void *value);
```

6.2.1.17 pthread__unlock_global_np()

POSIX 1003.4a Draft 4

```
void pthread_unlock_global_np();
```

POSIX Standard 1003.1c

```
int pthread_unlock_global_np(void);
```

6.2.2 Return Values

The new POSIX Threads Library POSIX 1003.1c interface does not use `errno`. (Note that POSIX Threads Library still provides a thread-specific `errno` cell for use by libraries and application code, but the 1003.1c interface does not write to this cell.) If an error condition occurs, a pthread routine returns an integer value indicating the type of error. For example, a call to the Draft 4 implementation of `pthread_cond_destroy` that returned a -1 and set `errno` to `EBUSY` now returns `EBUSY` as the routine return value.

On successful completion, most pthread routines return a zero.

The names and syntax of the following routines in this section did not change between 1003.4a draft and 1003.1c, but the return values are different. See the reference pages for details.

6.2.2.1 pthread_attr_setinheritsched()

POSIX 1003.4a Draft 4

[EINVAL]

The value specified by `attr` is invalid.

[EINVAL]

The value specified by `inherit` is invalid.

POSIX Standard 1003.1c

[EINVAL]

One or both of the values specified by `inherit` or by `attr` is invalid.

[ENOTSUP]

An attempt was made to set the attribute to an unsupported value.

6.2.2.2 pthread_cond_timedwait()

POSIX 1003.4a Draft 4

[EINVAL]

The value specified by `cond`, `mutex`, or `abstime` is invalid.

[EAGAIN]

The time specified by `abstime` expired.

[EDEADLK]

A deadlock condition is detected.

POSIX Standard 1003.1c

[EINVAL]

The value specified by `cond`, `mutex`, or `abstime` is invalid, or: Different mutexes are supplied for concurrent `pthread_cond_timedwait` operations or `pthread_cond_wait` operations on the same condition variable, or: The `mutex` was not owned by the current thread at the time of the call.

[ETIMEDOUT]

The time specified by `abstime` expired.

[ENOMEM]

POSIX Threads Library cannot acquire memory needed to block using a statically initialized condition variable.

6.2.2.3 `pthread_cond_wait()`

POSIX 1003.4a Draft 4

[EINVAL]

The value specified by `cond` or `mutex` is invalid.

[EDEADLK]

A deadlock condition is detected.

POSIX Standard 1003.1c

[EINVAL]

The value specified by `cond`, or `mutex` is invalid, or: Different mutexes are supplied for concurrent `pthread_cond_timedwait` operations or `pthread_cond_wait` operations on the same condition variable, or: The `mutex` was not owned by the current thread at the time of the call.

[ENOMEM]

POSIX Threads Library cannot acquire memory needed to block using statically initialized condition variable.

6.2.2.4 `pthread_mutex_trylock()`

POSIX 1003.4a Draft 4

1

Successful completion

0

The `mutex` is locked; therefore, it was not acquired.

-1 [EINVAL]

The value specified by `mutex` is invalid.

POSIX Standard 1003.1c

0

Successful completion

[EBUSY]

The `mutex` is locked; therefore, it was not acquired.

[EINVAL]

The value specified by `mutex` is invalid, or the `mutex` was created with the protocol attribute set to `PTHREAD_PRIO_PROTECT` and the calling thread's priority set higher than the `mutex`'s current priority ceiling.

6.2.2.5 `pthread_mutex_unlock()`

POSIX 1003.4a Draft 4

-1 [EINVAL]

The value specified by `mutex` is invalid.

POSIX Standard 1003.1c

[EINVAL]

The value specified for `mutex` is invalid.

[EPERM]

The calling thread does not own the `mutex`.

6.2.3 Unchanged Routines

The following threads routines are unchanged between POSIX 1003.4a draft 4 and POSIX 1003.1c:

```
pthread_self( )  
pthread_tescancel( )
```

6.2.4 New Routines

The following are POSIX 1003.1c pthread routines that did not exist in the 1003.4a Draft 4 implementation:

- `pthread_atfork()`
Declares handlers to be called when the process forks a child.
- `pthread_getconcurrency()`
Obtains the value of the concurrency-level global variable for the process.
- `pthread_setconcurrency()`
Changes the value of the concurrency-level global variable for the process. Because POSIX Threads Library automatically manages the concurrency of all threads in a multithreaded process, POSIX Threads Library ignores this concurrency level value. The routine is provided only for compatibility and has no effect on multithreaded programs that use POSIX Threads Library.
- `pthread_attr_getdetachstate()`
Obtains the `detachstate` attribute of the specified thread attributes object.
- `pthread_attr_setdetachstate()`
Changes the `detachstate` attribute of the specified thread attributes object.
- `pthread_key_delete()`
Deletes a thread-specific data key.
- `pthread_kill()`
Delivers a signal to a specified thread.
- `pthread_sigmask()`
Examines or changes the current thread's signal mask.
- `pthread_attr_getguardsize_np()`
Nonportable. Obtains the `guardsize` attribute of the specified thread attributes object.
- `pthread_attr_setguardsize_np()`
Nonportable. Changes the `guardsize` attribute of the specified thread attributes object.
- `pthread_cond_signal_int_np()`
Nonportable. Called only from the interrupt level. Wakes one thread that is waiting on the specified condition variable.
- `pthread_exc_get_status_np()`
Nonportable. Obtains a system-defined error status from a POSIX Threads Library status exception object.
- `pthread_exc_matches_np()`
Nonportable macro. Determines whether two POSIX Threads Library exception objects are identical.
- `pthread_exc_report_np()`

Nonportable. Produces a message that reports what a specified POSIX Threads Library status exception object represents.

- `pthread_exc_set_status_np()`

Nonportable macro. Imports a system-defined error status into a POSIX Threads Library address exception object.

- `pthread_getsequence_np()`

Nonportable. Obtains the unique identifier for the specified thread.

6.2.5 Routines with No Equivalent

The nonportable routine `pthread_signal_to_cancel_np()` has no equivalent in POSIX threads.

6.3 Porting Applications That Use UnixWare Threads

If the application you are porting employs UnixWare threads routines, you can often find a POSIX Threads Library routine that provides similar functionality. However a number of important differences exist between the UnixWare threads API and POSIX threads.

6.3.1 Porting Applications That Use Read-Write Locks

Starting with Tru64 UNIX Version 4.0F, the POSIX Threads Library implements read-write locks using the following routines. This implementation conforms to the UNIX 98 specification.

- `pthread_rwlock_destroy()`
- `pthread_rwlock_init()`
- `pthread_rwlock_rdlock()`
- `pthread_rwlock_tryrdlock()`
- `pthread_rwlock_trywrlock()`
- `pthread_rwlock_unlock()`
- `pthread_rwlock_wrlock()`
- `pthread_rwlockattr_destroy()`
- `pthread_rwlockattr_getpshared()`
- `pthread_rwlockattr_init()`
- `pthread_rwlockattr_setpshared()`

6.3.2 Features in UnixWare Threads Not in the POSIX Standard

The following features of UnixWare threads are not found in POSIX Threads Library:

- A thread's behavior cannot be specified to be like the behavior of a daemon. The `thr_create()` routine with the `daemon` attribute is not implemented in POSIX threads.
- Thread suspension and continuation is not supported. The following routines are not implemented in POSIX Standard 1003.1a:

```
thr_continue()  
thr_suspend()
```

The UnixWare documentation recommends against using these routines, so the issue might not even come up.

- Setting concurrency (requesting a new LWP) and determining concurrency level are not implemented in POSIX Standard 1003.1a. However, to provide for Single UNIX Specification, Version 2 source code compatibility, the POSIX Threads Library offers `pthread_getconcurrency()` and `pthread_setconcurrency()`. These routines offer functionality similar to the UnixWare routines `thr_getconcurrency()` and `thr_setconcurrency()`.

POSIX Threads Library automatically manages the concurrency of all threads in a multithreaded process and so ignores the value of the concurrency level. The concurrency level value has no effect on the behavior of a multithreaded program that uses POSIX Threads Library.

6.3.3 Features in the POSIX Standard Not in UnixWare Threads

POSIX threads offer attribute objects, thread cancellation, and thread scheduling, features that differ substantially from those in UnixWare threads.

6.3.3.1 Attributes Objects

Where UnixWare threads use option arguments to specify the state in which a thread entity is created, the approach in POSIX threads is to request the initialization state using an attributes object.

Attributes objects are a data type and are created and destroyed with one of the following routines:

- For thread attributes:

```
pthread_attr_init()
pthread_attr_destroy()
```

- For mutex attributes:

```
pthread_mutexattr_init()
pthread_mutexattr_destroy()
```

- For condition variable attributes:

```
pthread_condattr_init()
pthread_condattr_destroy()
```

When first created, attributes objects contain default values for the individual attributes. Attribute values are modified with one of the following routines:

- For thread attributes:

```
pthread_attr_setguardsize_np()
pthread_attr_setinheritsched()
pthread_attr_setschedpolicy()
pthread_attr_setstacksize()
```

- For mutex attributes:

```
pthread_mutexattr_settype_np()
```

- For condition variable attributes:

No routines are currently available for setting the attributes of condition variables. This will change in future releases.

To use a thread attributes object:

1. Create a thread attributes object by calling the `pthread_attr_init()` routine.
2. Call the appropriate routine to set the individual attributes of the thread attributes object.

3. Create a new thread by calling the `pthread_create()` routine and specifying the address of the thread attributes object.

Attributes objects offer improved portability and simplified state specification. Initialization is simple and localized, so future modifications can be made quickly and dependably.

A single attributes object can be used to create multiple objects. Use an attributes object in the same way that you use a class template.

When an attributes object is initialized, memory may be allocated for it. To return this memory to the system, use the appropriate `_destroy` routine.

6.3.3.2 Thread Cancellation

When a set of related threads is no longer needed, often the desirable action is to simply cancel the threads. For example, suppose a number of threads are working on the same task in parallel and one thread completes the task first. In this case, it might be desirable to cancel the remaining related threads.

Cancellations can occur under three circumstances:

- Asynchronously
- At points in the execution sequence as defined by the standard
- At cancellation points specified by the application

Cancellation tells the target thread to terminate as soon as possible. However, the target thread can control when it receives the request by controlling its cancelability state and type.

Initially a thread's cancelability state is enabled. The cancelability state determines whether a thread can receive a cancellation request. If the cancelability state is disabled, the thread does not receive any cancellation requests. You can change the current thread's cancelability state by calling `pthread_setcancelstate()`.

Initially a thread's cancelability type is deferred. Deferred cancelability means that a thread receives a cancellation request only at cancellation points, for example, when a call to `pthread_cond_wait()` is made. You can change the current thread's cancelability type by calling `pthread_setcanceltype()`.

The following is a list of routines that are cancellation points:

- `pthread_testcancel()`
- `pthread_delay_np()`
- `pthread_join()`
- `pthread_cond_wait()`
- `pthread_cond_timedwait()`
- POSIX blocking system calls

In addition to these POSIX cancellation points, critical non-POSIX functions such as `select()` are also defined as cancellation points. A list of these current and future system calls appears in Appendix A of the *Guide to the POSIX Threads Library*.

If a thread's cancelability type is asynchronous, the thread can receive a cancellation request at any time.

Use asynchronous cancellation carefully. In general, if asynchronous cancellation is enabled, it is prudent to follow these policies:

- Cancel only code you wrote or are similarly familiar with.

- Do not cancel POSIX Threads Library except to disable them.
- Do not cancel system calls or library functions.

After you cancel threads, restore resources and state with the POSIX functions `pthread_cleanup_push()` and `pthread_cleanup_pop()`.

6.3.3.3 Scheduling

POSIX threads are scheduled according to two factors:

- The scheduling policy
- The scheduling priority for the thread

Scheduling Policy

Three scheduling policies are available in POSIX Threads Library:

- `SCHED_FIFO` (first-in/first-out)

The highest-priority thread runs until it blocks. `SCHED_FIFO` is a POSIX portable policy.

- `SCHED_RR` (round-robin)

The highest-priority thread runs until it blocks; however, threads of equal priority are timesliced. `SCHED_RR` is a POSIX portable policy.

- `SCHED_OTHER`

The syntax for `SCHED_OTHER` is portable, but the POSIX specification allows each vendor to define the actual policy as the vendor chooses. So `SCHED_OTHER` on a UnixWare system results in a different scheduling policy than `SCHED_OTHER` in POSIX Threads Library.

In POSIX Threads Library, `SCHED_OTHER` implements the same policy as `SCHED_FG_NP` (foreground, nonportable). All threads are timesliced; scheduling is "starvation free." That is, even a compute-bound foreground thread does not prevent a low-priority background thread from running.

A second nonportable scheduling policy is available: `SCHED_BG_NP` (background, nonportable) As with `SCHED_FG_NP`, scheduling is starvation free. However, background threads receive less execution time than `SCHED_FG_NP` threads with the same priority.

The contention scope of a thread determines how it competes with other threads for processor resources. There are two contention scopes:

- `PTHREAD_SCOPE_SYSTEM`

All threads in system contention scope compete for processor resources against all other threads in system contention scope on the system, regardless of what process they are in.

- `PTHREAD_SCOPE_PROCESS`

All threads in process contention scope compete for processor resources only against other threads in the same process. When the process obtains processor resources, scheduling policy and priority determine which thread runs.

Threads in other processes whose state is also `PTHREAD_SCOPE_PROCESS` are scheduled independently.

In Tru64 UNIX, threads with different scope states can coexist on the same system and even in the same process.

Creating a POSIX Threads Library thread in the `PTHREAD_SCOPE_SYSTEM` state is equivalent to creating a UnixWare thread in the `THR_BOUND` state.

Scheduling Priority

Scheduling priority is expressed relative to other threads in the same policy in a range between a minimum and maximum for that scheduling policy.

Priority values are integers. Use arithmetic expressions to define values between the minimum and maximum priority. Because the range of priorities can vary between implementations, to avoid portability problems do not use specific numerical values. Use `sched_get_priority_min()` and `sched_get_priority_max()` to learn the minimum and maximum priorities of a scheduling policy. Values outside the range of minimum to maximum result in an error.

Keep in mind that scheduling is not the same as synchronization. Do not assume that a high-priority thread can access shared data without interference from low-priority threads. Two threads might run at the same time even though one of them is a FIFO-policy thread at the highest priority and the other is of background policy with lowest priority. Even on a four-processor system, do not assume that the four highest-priority threads will be executing at any given time.

Inheritance of Scheduling Attributes

The scheduling attributes of a newly created thread can either be inherited from the creating thread (the default) or they can be taken from the scheduling attributes stored in the attributes object.

The inheritance scheduling attribute determines which source of scheduling attributes is used. You set the inheritance scheduling attribute by calling the routine `pthread_attr_setinheritsched()`.

6.3.4 Synchronizing POSIX Threads

In POSIX Threads Library, thread synchronization can be achieved using the following:

- mutexes

Use mutexes to protect objects being accessed and control how threads share resources. A mutex allows a thread to lock shared data while using that data, so other threads do not interfere. Relevant pthreads routines include:

```
pthread_mutexattr_destroy( )
pthread_mutexattr_gettype( )
pthread_mutexattr_init( )
pthread_mutexattr_settype( )
pthread_mutex_destroy( )
pthread_mutex_init( )
pthread_mutex_lock( )
pthread_mutex_trylock( )
pthread_mutex_unlock( )
```

- condition variables

Use condition variables for communication. A condition variable can signal to a thread that shared data has reached some desired state. Relevant pthreads routines include:

```
pthread_condattr_destroy( )
pthread_condattr_init( )
pthread_cond_broadcast( )
pthread_cond_destroy( )
pthread_cond_init( )
pthread_cond_signal( )
```

```
pthread_cond_timedwait( )
pthread_cond_wait( )
```

Other Tru64 UNIX tools for synchronization are semaphores, pipes, and message queues.

6.3.5 UnixWare Threads Routines and POSIX Threads Routines

Table 6–2 maps UnixWare threads to the POSIX threads routines of Tru64 UNIX. See the appropriate reference page for information about particular POSIX Threads Library routines.

Table 6–2: UnixWare and POSIX Threads Library Routines

UnixWare Routine	Corresponding POSIX Routine
cond_broadcast()	pthread_cond_broadcast()
cond_destroy()	pthread_cond_destroy()
cond_init()	pthread_cond_init()
cond_signal()	pthread_cond_signal()
cond_timedwait()	pthread_cond_timedwait()
cond_wait()	pthread_cond_wait()
mutex_destroy()	pthread_mutex_destroy()
mutex_init()	thread_mutex_init()
mutex_lock()	pthread_mutex_lock()
mutex_trylock()	pthread_mutex_trylock()
mutex_unlock()	pthread_mutex_unlock()
thr_continue()	No equivalent
thr_create()	pthread_create()
thr_exit()	pthread_exit()
thr_getconcurrency()	pthread_getconcurrency()
thr_getprio()	thread_getschedparam()
thr_getspecific()	pthread_getspecific()
thr_join()	pthread_join()
thr_keycreate()	pthread_key_create()
thr_kill()	pthread_kill()
thr_min_stack()	Use PTHREAD_STACK_MIN, as defined in pthread.h
thr_self()	pthread_self()
thr_setconcurrency()	pthread_setconcurrency()
thr_setprio()	pthread_setschedparam()
thr_setspecific()	pthread_setspecific()
thr_sigsetmask()	pthread_sigmask()
thr_suspend()	No equivalent
thr_yield()	sched_yield()

6.3.6 For More Information

For detailed information about POSIX pthread functions in Tru64 UNIX, see the reference pages for specific pthread functions and the *Guide to the POSIX Threads Library* (formerly *Guide to DECthreads*).

6.4 Semaphores

Semaphores are not part of a threads implementation, but they do provide protected access to shared resources. Because of this, they are often treated as a topic related to threads. Tru64 UNIX implements the POSIX 1003.1b semaphore routines. UnixWare supports both the POSIX 1003.1b routines and a proprietary semaphore API. Table 6–3 maps the UnixWare routines to the POSIX 1003.1b API implemented in Tru64 UNIX.

Table 6–3: Semaphore Routines

UnixWare Semaphore Routines	Corresponding Tru64 UNIX POSIX 1003.1b Semaphore Routines
<code>sema_destroy()</code>	<code>sem_destroy()</code>
<code>sema_init()</code>	<code>sem_init()</code>
<code>sema_post()</code>	<code>sem_post()</code>
<code>sema_trywait()</code>	<code>sem_trywait()</code>
<code>sema_wait()</code>	<code>sem_wait()</code>

6.5 Programming Notes

This section presents issues related to programming with the POSIX Threads Library.

6.5.1 Assumptions About Deadlock Conditions

Although the POSIX Threads Library does not currently detect deadlock conditions involving more than one mutex, it might in the future. Therefore, avoid writing code that depends on the POSIX Threads Library not reporting a particular error condition.

6.5.2 Memory Alignment Considerations

Alpha processors prior to EV56 (EV4 and EV5) can access memory only in units of at least a longword (4 bytes). A longword in memory can contain multiple variables, each of which is less than 4 bytes. For these older processors, when a program accesses any part of a longword that contains more than one variable, it actually retrieves and writes the entire longword. In such cases, multithreaded programs can experience a problem if two or more threads read the same longword, update different parts of it, then independently write their respective copies back to memory. The last thread to write the longword overwrites any data previously written to other parts of the longword. This can happen even though each thread protects its part of the longword with its own mutex.

The Compaq C compiler protects scalar variables against this problem by aligning them in memory on longword (4-byte) boundaries. However, in composite data objects, such as structures or arrays, the compiler aligns members on their natural boundaries. For example, a 2-byte member is aligned on a 2-byte boundary. Because of this, any adjacent members of the composite object that total 4 bytes or less could occupy the same longword in memory.

Inspect your multithreaded application code to determine if you have a composite data object in which adjacent members could share the same longword in memory. If you do and if your project allows, we recommend that you force alignment of each such member variable to a longword boundary by redefining the variable to be at least 4 bytes, or by defining sufficient padding storage after the variable to total 4 bytes.

Alternatively, you can create one mutex for each composite data object in which adjacent members can share the same longword in memory. Then use this single mutex to protect all write accesses by all threads to the composite data object. This technique might be less desirable because of performance considerations.

To learn the type of Alpha processor on your system, use the following command:

```
/usr/sbin/psrinfo -v
```

For more information on threads and memory alignment, see the Granularity Considerations section of the *Guide to the POSIX Threads Library*.

6.5.3 POSIX Threads Library Extensions

In general, pthread routines ending in `_np` are not portable. The following Tru64 UNIX pthread routines are nonportable extensions and are not found in SCO UNIX. The object naming routines, `pthread_*name_np()`, are available in Tru64 UNIX Version 4.0F and higher releases. In the following list, they are marked with an asterisk.

```
pthread_attr_getguardsize_np( )
pthread_attr_getname_np( )*
pthread_attr_setguardsize_np( )
pthread_attr_setname_np( )*
pthread_cond_getname_np( )*
pthread_cond_setname_np( )*
pthread_cond_signal_int_np( )
pthread_delay_np( )
pthread_exc_get_status_np( )
pthread_exc_matches_np( )
pthread_exc_report_np( )
pthread_exc_set_status_np( )
pthread_get_expiration_np( )
pthread_getname_np( )*
pthread_getsequence_np( )
pthread_key_getname_np( )*
pthread_key_setname_np( )*
pthread_lock_global_np( )
pthread_mutex_getname_np( )*
pthread_mutex_setname_np( )*
pthread_mutexattr_gettype_np( )
pthread_mutexattr_settype_np( )
pthread_rwlock_getname_np( )*
pthread_rwlock_setname_np( )*
pthread_setname_np( )*
pthread_unlock_global_np( )
```

6.5.4 Fork Handlers

The `fork()` function in Tru64 UNIX creates a new process (child process) that is identical to the calling process (parent process). The `pthread_atfork()` routine allows a main program or library to control resources during a `fork()` operation. You can use the `pthread_atfork()` routine to ensure that program context in the child process is consistent and meaningful. For more information, see `pthread_atfork(3)` and `fork(2)`.

6.5.5 Thread Local Storage

Thread Local Storage (TLS) is data that has static extent (that is, not on the stack) for the lifetime of a thread in a multithreaded process, and whose allocation is

specific to each thread. In standard multithreaded programs, static-extent data is shared among all threads of a given process, whereas thread local storage is allocated on a per-thread basis such that each thread has its own copy of the data that can be modified by that thread without affecting the value seen by the other threads in the process.

Thread Local Storage (TLS) support is always enabled in the C compiler. The `cc` command's `-ms` option is not required. In C++, TLS is recognized only with the `-ms` option, and it is otherwise treated as an error.

For more information, see the *Guide to the POSIX Threads Library*.

6.5.6 Thread-Independent Services

Thread-independent services (TIS) is a proprietary interface of the Tru64 UNIX operating system. You use `tis` routines to build thread-safe code libraries whose routines can be called from either a single-threaded or a multithreaded environment. In the absence of threads, `tis` routines impose minimal overhead on the calling program.

For instance, `tis` routines avoid the use of interlocked instructions and memory barriers. When threads are present, `tis` routines provide full support for POSIX threads synchronization. However, there are no `tis` routines for creating threads or thread objects, because that would have no meaning if called from a single-threaded environment.

6.6 Files

Threads on Tru64 UNIX systems use the include files and shared libraries described in this section.

6.6.1 Include Files

Threads on Tru64 UNIX systems use the following include files:

- `pthread.h`
Provides the POSIX 1003.1c–1995 API. Include the `pthread.h` file in your program. For more information, see the comments in the `pthread.h` file.
- `pthread_exception.h`
Supports `pthread` exception handling. Each C module that uses the POSIX Threads Library exceptions must include the `pthread_exception.h` header file. The *Guide to the POSIX Threads Library* describes the use of POSIX Threads Library exceptions.

6.6.2 Shared Libraries

Threads on Tru64 UNIX systems use the shared libraries described in Table 6–4.

Table 6–4: Shared Libraries

Library	Description
<code>libmach.so</code>	Shared version of threads support library. Direct use of <code>mach</code> interfaces is not supported.
<code>libpthread.so</code>	Requires <code>libmach.so</code> , <code>libexc.so</code> , and <code>libc.so</code> .

Table 6–4: Shared Libraries (cont.)

Library	Description
libexc.so	Shared version of Tru64 UNIX exception support package.
libc.so	Shared version of libc (C language runtime) package.

The `libexc`, `libmach`, `libpthread`, and `libc` libraries are available to code compiled with the `-pthread` option to the C compiler command. For example:

```
# cc -o myprog myprog.c -pthread
```

6.7 Linking

The `ld` command does not support the `-pthread` switch or the `-threads` switch. Instead, you must list the individual libraries in the proper order. For example:

```
ld <...> -lpthread -lmach -lexc -lc
```

More generally, link a multithreaded application with the following switches:

```
-lpthread -lmach -lexc -lc
```

6.8 Compiling

Compile a multithreaded application by using shared versions of `libmach` and `libpthread` as follows:

```
# cc -o myprog myprog.c -pthread
```

If you use a compiler front end or a language environment that does not support the `-pthread` compilation switch, you must use the `-D_THREAD_SAFE` compilation switch.

6.9 Debugging

Tru64 UNIX offers three threads debugging tools for multithreaded applications:

- Ladebug debugger
- Visual Threads
- ATOM

6.9.1 Ladebug Debugger

The Ladebug debugger provides commands to display the state of threads, mutexes, and condition variables, without using the built-in POSIX Threads Library debug facility. Using the Ladebug commands, you can examine core files and remote debug sessions, as well as run processes.

The following list is a summary of Ladebug threads commands:

- Set breakpoints:

```
stop [variable] [thread IDs] [at line] [if expression]
```

```
stop [variable] [thread IDs] [in function] [if expression]
```

IDs is a list of thread IDs. If you omit *IDs*, the breakpoint is set at the process level.

- Set tracepoints:

```
trace [variable] [thread IDs] [at line] [if expression]
```

```
trace [variable] [thread IDs] [in function] [if expression]
```

```
when [variable] [thread IDs] [at line] [if expression]  
{command [; . . .]}
```

```
when [variable] [thread IDs] [at line] [in function]  
{command [; . . .]}
```

If you do not specify *IDs*, the tracepoint is set at the process level.

- Step through the current thread:

```
step  
stepi  
next  
nexti
```

- Resume execution of a thread put on hold, for example, at a breakpoint:

```
cont [signal]
```

As the current thread resumes, all other threads also continue.

If you specify a signal, the program continues execution with that signal. The signal value can be either a signal number or a string name (for example, SIGSEGV).

- Show threads known to the debugger:

```
show thread [IDs][with state == state_specification]
```

If you do not specify any thread identifiers, the debugger displays information for all known threads.

To list threads in a specific state, such as threads that are currently blocked, use the option `show thread with state ==`. For example:

```
show thread [IDs] with state == blocked
```

The following are valid values for *state_specification*:

```
ready  
blocked  
running  
terminated  
detached
```

- Display the stack trace of current thread:

```
where [number] [thread IDs | all | * ]
```

The command `where [number]` displays the stack trace of the currently active functions for the current thread.

The command `where [thread]` displays the stack traces of the specified threads.

The qualifiers `all` and `*` are equivalent.

- Show the registers for the current thread:

```
printreg
```

- Evaluate an expression in the context of the current thread:

```
print expression
```

- Evaluate an expression in the context of the current thread and make a call in that context:

```
call expression
```

- Show information about mutexes:

```
show mutex [mutex_identifier_list] [with state == locked]
```

Use the optional `state == locked` to display information only for locked mutexes.

- Show information about condition variables:

```
show condition [condition_IDs] [with state == wait]
```

If you supply one or more condition variable IDs, the debugger displays information about those condition variables that you specify, provided that the list matches the identity of currently available condition variables. If you omit the condition variable identifier specification, the debugger displays information about all condition variables currently available.

For details and examples, see the *Ladebug Debugger Manual*.

6.9.2 Visual Threads

The Visual Threads tool lets you debug and analyze Tru64 UNIX multithreaded applications that use the POSIX Threads Library or that are written in Java. You can use it to debug potential thread-related logic problems, including deadlock, data protection errors, and thread usage errors. You can also use its rule-based analysis and statistics capabilities to monitor the thread-related performance of an application. It can help you identify problem areas even though the application does not show any specific problem symptoms.

Visual Threads is available on the Associated Products CD-ROM for Tru64 UNIX and is licensed as part of the Developers' Toolkit. You can also download Visual Threads from the Visual Threads Web site at the following location:

```
http://www.tru64unix.compaq.com/visualthreads/
```

This site also contains information about installing, licensing, starting, and using Visual Threads. Visual Threads is documented in online help.

6.9.3 ATOM

You can obtain functionality similar to that of the `truss` debugger by using ATOM, which supports debugging multithreaded programs.

ATOM provides a mechanism for instrumenting an application by collecting and analyzing a wide variety of data. ATOM contains a prepackaged trace tool that prints the name of each procedure as it is called.

To use the ATOM trace tool, do the following:

1. Compile your application with the option `-g1`, which produces symbol-table information:

```
# cc -g1 app_source -o app_file
```
2. Build an instrumented version of your application (ATOM creates a file named `app_object.ptrace`):

```
# atom app_file -tool ptrace
```
3. Execute `app_file.ptrace` and view the output.

To instrument the shared libraries and system shared libraries of an application, use either the `-all` or the `-incobj` option to ATOM. The `-ldir` option causes ATOM to search the specified directory for shared libraries. This option is useful when an application stores its shared libraries in a separate location from the executables.

The instrumented shared libraries are placed in the current directory unless you use the `-shlibdir` option is used to specify a different directory. When running

the instrumented application, you must set the `LD_LIBRARY_PATH` environment variable to point to the directory containing the instrumented shared libraries.

Transferring Data

If data with multibyte values is being transferred between big-endian and little-endian systems, then you must provide code that swaps the byte order. This is a simple matter. For 32-bit data, code to convert big-endian to little-endian data might look as follows:

```
#define SWAP4(N)      (      (( (N) & 0x00ff)   << 24) | \
                          (( (N) & 0xff00)   <<  8) | \
                          (( (N) & 0xff0000) >>  8) | \
                          (( (N) & 0xff000000) >> 24)  \
                          )
```

The following I/O is transparent to endianism:

- Data that is read and written by the framework classes
The framework classes always store data on disk in a type-tagged, compressed binary format. The processing is such that either machine kind can read or write data. The on-disk format is the same for both types of machines.
- Temporary files that are written then read by the same invocation of an application and that are deleted before the application terminates

If you have big-endian applications that have direct written persistent data, you can update the readers/writers in `LITTLE_ENDIAN` conditionally compiled code to do the byte reversal on read/write.

Suggestion: If your data file has a versioning indicator, define the existing version to be big endian. Invent a new version value to mean little endian. Make the `READER` able to read both kinds of files. The big-endian `WRITER` is unchanged, and the little-endian `WRITER` writes the little-endian version. This approach has the following advantages:

- A write takes minimal time
- Reads on the same machine type take (near) minimal time
- Only cross-endian reads take more time (assumed to be the rare case)

A.1 Using Fortran to Convert Unformatted Numeric Data

Compaq Visual Fortran enables programs to read and write unformatted data (originally written using unformatted I/O statements) in several nonnative floating-point formats and in big-endian `INTEGER` or floating-point format. This facilitates sharing a common source of unformatted data among big-endian and little-endian systems.

Converting unformatted data rather than formatted data is generally faster and is less likely to lose precision of floating-point numbers. If a converted nonnative value is outside the range of the native data type, a run-time message appears. (See the Compaq Visual Fortran documentation for a list of run-time messages.)

Table A-1 lists the keywords for some of the supported unformatted file data formats. The Compaq Visual Fortran documentation presents other keywords you can use.

Table A–1: Data Conversion Keywords

Keyword	Results
<code>little_endian</code>	Native little-endian integers of the appropriate INTEGER size (1, 2, 4, or 8 bytes) and native little-endian IEEE floating-point data for REAL and COMPLEX single-precision and double-precision numbers. These are the same formats as stored in memory.
<code>big_endian</code>	Big-endian integer data of the appropriate INTEGER size (1, 2, or 4 bytes) and big-endian IEEE floating-point formats for REAL and COMPLEX single-precision and double-precision numbers. INTEGER (KIND=1) or INTEGER*1 data is the same for little endian and big endian.
<code>native</code>	No conversion occurs between memory and disk. This is the default for unformatted files.

You can use the following methods to specify the type of nonnative (or native) format:

- Set an environment variable for a specific unit number before the file is opened. The environment variable is named `FORT_CONVERT n` , where n is the unit number.

You can use environment variables to specify multiple formats in a single program, usually one format for each specified unit number. Specify the numeric format at run time by setting the appropriate environment variable before you open that unit number.

When you open a file, the appropriate environment variable is set. That environment variable is always used. For example, you might use this method to specify that a unit number will use a particular format instead of the format specified in the program (perhaps in a script file that sets the environment variable before running the program).

Suppose you have previously compiled a program that reads numeric data from unit 28. The following Korn shell command sequence sets the appropriate environment variables before running the program:

```
# FORT_CONVERT28 = little_endian
# export FORT_CONVERT28
```

- Specify the `CONVERT` specifier in the `OPEN` statement for a specific unit number.

For example, the following source code shows how the `OPEN` statement would be coded to read nonnative big-endian (IEEE floating-point) numeric data from unit 15. This data might be processed and written in native little-endian format to unit 20 (the absence of the `CONVERT` specifier or environment variable `FORT_CONVERT20` indicates native Alpha little-endian data for unit 20):

```
OPEN (CONVERT='big_endian', FILE='graph3.dat',
FORM='UNFORMATTED', UNIT=15)
.
.
.
OPEN (FILE='graph3_axp.dat', FORM='UNFORMATTED', UNIT=20)
```

- Compile the program with the `f77 -convert` command. This method affects all unit numbers that use unformatted data specified by the program.

Specify the numeric format at compile time. You must compile all routines under the same `-convert` option. You could use the same source program and compile it using different `f77` commands to create multiple executable programs, each of which reads and writes different formats.

For example, the following shell commands compile program `file.f` to use big-endian integer data and big-endian IEEE floating-point formats. Data is converted between the big-endian format and the little-endian memory format

(little-endian integers, `S_float` and `T_float` little-endian IEEE floating-point format).

```
# f77 -convert big_endian -o big_endian.out file.f
$# big_endian.out
```

Because this method affects all unit numbers, you cannot read data in one format and write it in another file format unless you also use it in combination with the environment variable method or the `OPEN` statement `CONVERT` specifier method to specify a different format for a particular unit number.

If you specify more than one method, the order of precedence when you open a file with unformatted data is to look for an environment variable first, then for the `OPEN` statement `CONVERT` specifier, and then whether the `f77 -convert` command was used when the program was compiled.

A.2 Nonnative Data

When you port source code along with the unformatted data, be aware that vendors might use different units for specifying the record length (`RECL` specifier) of unformatted files. Whereas formatted files are specified in units of characters (bytes), unformatted files are specified in longword units for Compaq Fortran and some other vendors.

Resources on the Internet

The following porting-related resources are available on the Internet:

- The Tru64 UNIX documentation library:
`http://www.tru64unix.compaq.com/faqs/publications/pub_page/pubs_page.html`
- The Tru64 UNIX *Software Product Description*:
`http://www.tru64unix.compaq.com/unix/technical.htm`
- Compaq Solutions Alliance (CSA):
`http://csa.compaq.com`
- Information about Tru64 UNIX and related products, services, and technical assistance:
`http://www.tru64unix.compaq.com`
- Compaq software archive (unsupported):
`http://ftp.digital.com`
- Porting Assistant (also bundled on the Tru64 UNIX CD-ROMS):
`http://www.tru64unix.compaq.com/info/portingassistant/`
- An unmoderated newsgroup on matters related to Compaq Computer Corporation:
`comp.sys.dec`
- UnixWare Version 7 documentation:
`http://doc.sco.com/`
- OpenServer documentation:
`http://osr5doc.sco.com:1996/dochome.html`

Numbers and Special Characters

32-bit applications

- constants, 2–10
- pointers, 4–4
- porting guidelines, 3–1
- variables, 2–11

32-bit considerations, 2–7

32-bit environment

- pointers, 2–10

64-bit considerations, 2–7

- constants, 2–10
- pointers, 4–4
- structures, 2–11
- variables, 2–11

64-bit environment

- pointers, 2–9

A

addresses, network, 3–2

archive libraries, 4–16

argument passing, 4–5

asynchronous cancellation of threads, 6–11

ATOM, 6–20

attribute objects

- pthreads, 6–10

B

big endian, A–1

bit fields, 2–13

bit shift operation, 2–11

C

C language, 4–2

compiler, 4–2

Kernighan and Ritchie (K&R) C, 4–2

misuse of functions, 4–12

preprocessor, 4–2

System V Habitat, 4–7

C++, 4–6

call conventions, 4–5

calloc, 2–13

CDSL, 2–7

Compaq Solutions Alliance (CSA),
1–2

compiler, 3–3, 4–12, 6–15, 6–18

C language, 4–2

C++, 4–6

options, 4–3

portability options, 4–3

-xtaso, 4–4

constants, 2–10

D

data

access, 2–8

alignment, 2–8, 3–2, 3–4

transfer, A–1

dbx, 3–5, 4–6

deadlock conditions, 6–15

debuggers, 4–6, 6–18

ATOM, 6–20

dbx, 4–6

Ladebug, 4–6, 6–18

Visual Threads, 6–18, 6–20

defines, 2–6

development tools, 2–2, 4–1

device naming, 2–6

diagnostic messages, 5–2

directories, 2–4

context-dependent symbolic links, 2–6

device naming, 2–6

E

endian, 2–14

interoperability, A–1

errno, 6–6

F

fgetpos (), 2–14

file systems, 2–14

float.h, 2–6

fork handlers, 6–16

fsetpos (), 2–14

functions, 3–5, 4–12, 5–5

G

grep, 3–2

H

header files, 4–11
HyperHelp, 5–2

I

include files, 6–17
inheritance, 6–13
interfaces, 4–3
Internet, URLs of resources, B–1
interoperability
 data transfer, A–1
 file systems, 2–14
 record lengths (RECL), A–3
 variables, 2–8

K

kdbx, 4–6
Kernighan and Ritchie (K&R) C, 4–2

L

Ladebug, 4–6, 6–18
library calls, 2–13
limits.h, 2–6
linker, 4–5, 6–18
lint, 4–6
 -Q option, 3–2, 3–3
little endian, A–1
longword, 2–9, 6–15
lseek system call, 2–13

M

macros, 4–3
 make command, 4–11
make command, 4–9
 directives, 4–10
 macros, 4–11
 options, 4–10
 rules, 4–10
makefiles, 4–9
malloc, 2–13
maximum and minimum numeric values, 2–6
member alignment, 2–11
memory, 2–14
 alignment, 6–15

N

network addresses, 3–2
numeric values, 2–6

O

options, 3–3
 compilation, 4–3
 make, 4–10
 -taso option, 3–4
 -xtaso, 4–4

P

padding, 2–12
pointers, 2–9, 3–1
 32-bit, 2–10, 4–4
 64-bit, 2–10, 4–4
 subtraction of, 2–10
 truncations, 4–5
porting, 1–1
 guidelines, 3–1
 problems, 3–2
 procedures, 3–2
 services, 1–2
 suggestions, 3–4
 threaded applications, 6–1
 tools, 1–2
Porting Assistant, 5–1
 diagnostic messages, 5–2
 functions, 5–1
 HyperHelp, 5–2
 limitations, 5–5
 performing checks, 5–3
 using, 5–3
printf function, 2–13
programming tools, 4–8
pthreads, 6–1

 asynchronous cancellation, 6–11
 attribute inheritance, 6–13
 attribute objects, 6–10
 extensions, 6–16
 include files, 6–17
 routines, 6–14
 scheduling, 6–12
 thread cancellation, 6–11
 UnixWare, 6–14

Q

quadword, 2–9

R

read-write locks, 6–9
real time programming, 2–3
RECL (record length specifiers), A–3
return values, 6–6
routines, 6–2
 new, 6–8
 unchanged, 6–8
 with no equivalent, 6–9

S

scanf function, 2-13
scheduling policy threads, 6-12
scheduling priority, 6-13
SCO UNIX preprocessor, 4-2
SCO UNIX software version, 1-1
search path
 makefile, 4-9
segmentation faults, 2-10
semaphores, 2-9, 6-15
shared libraries, 4-12, 6-17
 using, 4-16
size of operator, 2-12
software version, 1-1
structures, 2-11
 alignment, 2-12
subtraction of pointers, 2-10
support
 developer support, 1-2
symbolic links, 2-6
syntax changes, 6-2
System V
 compatibility, 4-8
 Habitat, 4-7
 Interface Definition (SVID), 4-8

T

taso option, 4-5
thread local storage, 6-16
thread-independent services, 6-17
threaded applications, 6-1
threads, 2-9, 6-1, 6-9
 cancellation, 6-11

 include files, 6-17
 priority values, 6-13
 routines, 6-14
 scheduling, 6-12
 semaphores, 6-15
 UnixWare, 6-9
transfer of data, A-1
Tru64 UNIX, 2-1
 compatibility, 4-7
 development environment, 4-1
 file system, 2-14
 programming tools, 4-8
truncation
 of longs, 2-10
 of pointers, 2-10

U

unformatted data, A-1
unions, 2-13
UnixWare threads and pthreads, 6-14

V

variables, 2-11
 environment, A-2
version
 software, 1-1
Visual Fortran, A-1
Visual Threads, 6-20

X

-xtaso compiler option, 3-4