

**IEEE Std 1003.1™, 2004 Edition**

**The Open Group Technical Standard  
Base Specifications, Issue 6**

**Includes IEEE Std 1003.1™-2001, IEEE Std 1003.1™-2001/Cor 1-2002  
and IEEE Std 1003.1™-2001/Cor 2-2004**

# **Standard for Information Technology— Portable Operating System Interface (POSIX®)**

## **Rationale (Informative)**

Sponsor

**Portable Applications Standards Committee  
of the  
IEEE Computer Society**

and

**The Open Group**



THE *Open* GROUP

[This page intentionally left blank]

## Abstract

This standard is simultaneously ISO/IEC 9945, IEEE Std 1003.1, and forms the core of the Single UNIX Specification, Version 3.

This 2004 Edition includes IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004 incorporated into IEEE Std 1003.1-2001 (the base document). The two Corrigenda address problems discovered since the approval of IEEE Std 1003.1-2001. These changes are mainly due to resolving integration issues raised by the merger of the base documents that were incorporated into IEEE Std 1003.1-2001, which is the single common revision to IEEE Std 1003.1<sup>™</sup>-1996, IEEE Std 1003.2<sup>™</sup>-1992, ISO/IEC 9945-1: 1996, ISO/IEC 9945-2: 1993, and the Base Specifications of The Open Group Single UNIX<sup>®</sup> Specification, Version 2.

This standard defines a standard operating system interface and environment, including a command interpreter (or “shell”), and common utility programs to support applications portability at the source code level. This standard is intended to be used by both applications developers and system implementors and comprises four major components (each in an associated volume):

- General terms, concepts, and interfaces common to all volumes of this standard, including utility conventions and C-language header definitions, are included in the Base Definitions volume.
- Definitions for system service functions and subroutines, language-specific system services for the C programming language, function issues, including portability, error handling, and error recovery, are included in the System Interfaces volume.
- Definitions for a standard source code-level interface to command interpretation services (a “shell”) and common utility programs for application programs are included in the Shell and Utilities volume.
- Extended rationale that did not fit well into the rest of the document structure, which contains historical information concerning the contents of this standard and why features were included or discarded by the standard developers, is included in the Rationale (Informative) volume.

The following areas are outside the scope of this standard:

- Graphics interfaces
- Database management system interfaces
- Record I/O considerations
- Object or binary code portability
- System configuration and resource availability

This standard describes the external characteristics and facilities that are of importance to applications developers, rather than the internal construction techniques employed to achieve these capabilities. Special emphasis is placed on those functions and facilities that are needed in a wide variety of commercial applications.

## Keywords

application program interface (API), argument, asynchronous, basic regular expression (BRE), batch job, batch system, built-in utility, byte, child, command language interpreter, CPU, extended regular expression (ERE), FIFO, file access control mechanism, input/output (I/O), job control, network, portable operating system interface (POSIX<sup>®</sup>), parent, shell, stream, string, synchronous, system, thread, X/Open System Interface (XSI)

Rationale (Informative)

Published 30 April 2004 by the Institute of Electrical and Electronics Engineers, Inc.  
3 Park Avenue, New York, NY 10016-5997, U.S.A.  
ISBN: 0-7381-4045-7/SH95237 PDF 0-7381-4046-5/SS95237  
Printed in the United States of America by the IEEE.

Published 30 April 2004 by The Open Group  
Apex Plaza, Forbury Road, Reading, Berkshire RG1 1AX, U.K.  
Document Number: C049  
ISBN: 1-931624-46-1  
Printed in the U.K. by The Open Group.

All rights reserved. No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without prior written permission from both the IEEE and The Open Group.

Portions of this standard are derived with permission from copyrighted material owned by Hewlett-Packard Company, International Business Machines Corporation, Novell Inc., The Open Software Foundation, and Sun Microsystems, Inc.

**Permissions**

Authorization to photocopy portions of this standard for internal or personal use is granted provided that the appropriate fee is paid to the Copyright Clearance Center or the equivalent body outside of the U.S. Permission to make multiple copies for educational purposes in the U.S. requires agreement and a license fee to be paid to the Copyright Clearance Center.

Beyond these provisions, permission to reproduce all or any part of this standard must be with the consent of both copyright holders and may be subject to a license fee. Both copyright holders will need to be satisfied that the other has granted permission. Requests to the copyright holders should be sent by email to [austin-group-permissions@opengroup.org](mailto:austin-group-permissions@opengroup.org).

**Feedback**

This standard has been prepared by the Austin Group. Feedback relating to the material contained in this standard may be submitted using the Austin Group web site at [www.opengroup.org/austin/defectform.html](http://www.opengroup.org/austin/defectform.html).

## IEEE

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. The IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and serve without compensation. While the IEEE administers the process and establishes rules to promote fairness in the consensus development process, the IEEE does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

Use of an IEEE Standard is wholly voluntary. The IEEE disclaims liability for any personal injury, property, or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon this, or any other IEEE Standard document.

**The IEEE does not warrant or represent the accuracy or content of the material contained herein, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or fitness for a specific purpose, or that the use of the material contained herein is free from patent infringement. IEEE Standards documents are supplied “AS IS”.**

The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

In publishing and making this document available, the IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is the IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEEE Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of the IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position, explanation, or interpretation of the IEEE. Current interpretations can be accessed at <http://standards.ieee.org/reading/ieee/interp/index.html>.

Errata, if any, for this and all other standards can be accessed at <http://standards.ieee.org/reading/ieee/updates/errata/index.html>. Users are encouraged to check this URL for errata periodically.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with the IEEE.<sup>1</sup> Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, U.S.A.

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents for which a license may be required by an IEEE Standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

A patent holder has filed a statement of assurance that it will grant licenses under these rights without compensation or under reasonable rates and non-discriminatory, reasonable terms and conditions to all applicants desiring to obtain such licenses. The IEEE makes no representation as to the reasonableness of rates and/or terms and conditions of the license agreements offered by patent holders. Further information may be obtained from the IEEE Standards Department.

Authorization to photocopy portions of any individual standard for internal or personal use is granted in the U.S. by the Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to the Copyright Clearance Center.<sup>2</sup> Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center. To arrange for payment of the licensing fee, please contact:

Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923, U.S.A., Tel.: +1 978 750 8400

Amendments, corrigenda, and interpretations for this standard, or information about the IEEE standards development process, may be found at <http://standards.ieee.org>.

The IEEE publications catalog and ordering information are available at <http://shop.ieee.org/store>.

1. For this standard, please send comments via the Austin Group as requested on page ii.

2. Please refer to the special provisions for this standard on page ii concerning permissions from both copyright holders and arrangements to cover photocopying and reproduction across the world, as well as by commercial organizations wishing to license the material for use in product documentation.

## The Open Group

The Open Group is a vendor-neutral and technology-neutral consortium, whose vision of Boundaryless Information Flow will enable access to integrated information within and between enterprises based on open standards and global interoperability. The Open Group works with customers, suppliers, consortia, and other standards bodies. Its role is to capture, understand, and address current and emerging requirements, establish policies, and share best practices; to facilitate interoperability, develop consensus, and evolve and integrate specifications and Open Source technologies; to offer a comprehensive set of services to enhance the operational efficiency of consortia; and to operate the industry's premier certification service, including UNIX certification.

Further information on The Open Group is available at [www.opengroup.org](http://www.opengroup.org).

The Open Group has over 15 years' experience in developing and operating certification programs and has extensive experience developing and facilitating industry adoption of test suites used to validate conformance to an open standard or specification. The Open Group portfolio of test suites includes the *Westwood* family of tests for this standard and the associated certification program for Version 3 of the Single UNIX Specification, as well tests for CDE, CORBA, Motif, Linux, LDAP, POSIX.1, POSIX.2, POSIX Realtime, Sockets, UNIX, XPG4, XNFS, XTI, and X11. The Open Group test tools are essential for proper development and maintenance of standards-based products, ensuring conformance of products to industry-standard APIs, applications portability, and interoperability. In-depth testing identifies defects at the earliest possible point in the development cycle, saving costs in development and quality assurance.

More information is available at [www.opengroup.org/testing](http://www.opengroup.org/testing).

The Open Group publishes a wide range of technical documentation, the main part of which is focused on development of Technical and Product Standards and Guides, but which also includes white papers, technical studies, branding and testing documentation, and business titles. Full details and a catalog are available at [www.opengroup.org/pubs](http://www.opengroup.org/pubs).

As with all *live* documents, Technical Standards and Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards compatible and those which are not:

- A new *Version* indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it *replaces* the previous publication.
- A new *Issue* indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published at [www.opengroup.org/corrigenda](http://www.opengroup.org/corrigenda).

The Open Group publications catalog and ordering information are available at [www.opengroup.org/pubs](http://www.opengroup.org/pubs).

# Contents

<b>Part</b>	<b>A</b>	<b>Base Definitions.....</b>	<b>1</b>
<b>Appendix</b>	<b>A</b>	<b>Rationale for Base Definitions.....</b>	<b>3</b>
	A.1	Introduction .....	3
	A.1.1	Scope.....	3
	A.1.2	Conformance .....	5
	A.1.3	Normative References .....	5
	A.1.4	Terminology .....	5
	A.1.5	Portability .....	8
	A.1.5.1	Codes.....	8
	A.1.5.2	Margin Code Notation.....	9
	A.2	Conformance .....	9
	A.2.1	Implementation Conformance.....	9
	A.2.1.1	Requirements.....	9
	A.2.1.2	Documentation.....	9
	A.2.1.3	POSIX Conformance .....	10
	A.2.1.4	XSI Conformance .....	11
	A.2.1.5	Option Groups.....	11
	A.2.1.6	Options.....	12
	A.2.2	Application Conformance.....	12
	A.2.2.1	Strictly Conforming POSIX Application.....	12
	A.2.2.2	Conforming POSIX Application.....	12
	A.2.2.3	Conforming POSIX Application Using Extensions .....	12
	A.2.2.4	Strictly Conforming XSI Application .....	13
	A.2.2.5	Conforming XSI Application Using Extensions.....	13
	A.2.3	Language-Dependent Services for the C Programming Language.....	13
	A.2.4	Other Language-Related Specifications.....	13
	A.3	Definitions .....	13
	A.4	General Concepts.....	34
	A.4.1	Concurrent Execution.....	34
	A.4.2	Directory Protection .....	34
	A.4.3	Extended Security Controls.....	35
	A.4.4	File Access Permissions .....	35
	A.4.5	File Hierarchy .....	35
	A.4.6	Filenames.....	35
	A.4.7	File Times Update.....	37
	A.4.8	Host and Network Byte Order.....	37
	A.4.9	Measurement of Execution Time.....	37
	A.4.10	Memory Synchronization.....	37
	A.4.11	Pathname Resolution .....	39
	A.4.12	Process ID Reuse .....	40
	A.4.13	Scheduling Policy.....	40

A.4.14	Seconds Since the Epoch .....	40
A.4.15	Semaphore.....	42
A.4.16	Thread-Safety.....	42
A.4.17	Tracing.....	42
A.4.18	Treatment of Error Conditions for Mathematical Functions .....	42
A.4.19	Treatment of NaN Arguments for Mathematical Functions .....	42
A.4.20	Utility.....	42
A.4.21	Variable Assignment .....	42
A.5	File Format Notation.....	42
A.6	Character Set.....	43
A.6.1	Portable Character Set.....	43
A.6.2	Character Encoding.....	43
A.6.3	C Language Wide-Character Codes .....	44
A.6.4	Character Set Description File.....	44
A.6.4.1	State-Dependent Character Encodings .....	44
A.7	Locale.....	46
A.7.1	General.....	46
A.7.2	POSIX Locale .....	47
A.7.3	Locale Definition .....	47
A.7.3.1	LC_CTYPE.....	48
A.7.3.2	LC_COLLATE.....	49
A.7.3.3	LC_MONETARY.....	51
A.7.3.4	LC_NUMERIC.....	52
A.7.3.5	LC_TIME.....	52
A.7.3.6	LC_MESSAGES.....	53
A.7.4	Locale Definition Grammar.....	53
A.7.4.1	Locale Lexical Conventions.....	53
A.7.4.2	Locale Grammar.....	53
A.7.5	Locale Definition Example.....	54
A.8	Environment Variables .....	57
A.8.1	Environment Variable Definition .....	57
A.8.2	Internationalization Variables.....	57
A.8.3	Other Environment Variables.....	58
A.9	Regular Expressions.....	60
A.9.1	Regular Expression Definitions .....	60
A.9.2	Regular Expression General Requirements.....	61
A.9.3	Basic Regular Expressions .....	62
A.9.3.1	BREs Matching a Single Character or Collating Element.....	62
A.9.3.2	BRE Ordinary Characters.....	62
A.9.3.3	BRE Special Characters.....	62
A.9.3.4	Periods in BREs.....	62
A.9.3.5	RE Bracket Expression .....	62
A.9.3.6	BREs Matching Multiple Characters.....	64
A.9.3.7	BRE Precedence .....	64
A.9.3.8	BRE Expression Anchoring.....	64
A.9.4	Extended Regular Expressions .....	65
A.9.4.1	EREs Matching a Single Character or Collating Element.....	65
A.9.4.2	ERE Ordinary Characters.....	65



A.9.4.3	ERE Special Characters.....	65
A.9.4.4	Periods in EREs.....	65
A.9.4.5	ERE Bracket Expression.....	65
A.9.4.6	EREs Matching Multiple Characters.....	65
A.9.4.7	ERE Alternation.....	65
A.9.4.8	ERE Precedence.....	66
A.9.4.9	ERE Expression Anchoring.....	66
A.9.5	Regular Expression Grammar.....	66
A.9.5.1	BRE/ERE Grammar Lexical Conventions.....	66
A.9.5.2	RE and Bracket Expression Grammar.....	66
A.9.5.3	ERE Grammar.....	66
A.10	Directory Structure and Devices .....	67
A.10.1	Directory Structure and Files .....	67
A.10.2	Output Devices and Terminal Types .....	67
A.11	General Terminal Interface .....	67
A.11.1	Interface Characteristics.....	68
A.11.1.1	Opening a Terminal Device File .....	68
A.11.1.2	Process Groups.....	68
A.11.1.3	The Controlling Terminal.....	69
A.11.1.4	Terminal Access Control .....	69
A.11.1.5	Input Processing and Reading Data.....	70
A.11.1.6	Canonical Mode Input Processing .....	70
A.11.1.7	Non-Canonical Mode Input Processing.....	71
A.11.1.8	Writing Data and Output Processing .....	71
A.11.1.9	Special Characters.....	71
A.11.1.10	Modem Disconnect.....	71
A.11.1.11	Closing a Terminal Device File .....	71
A.11.2	Parameters that Can be Set .....	72
A.11.2.1	The termios Structure .....	72
A.11.2.2	Input Modes.....	72
A.11.2.3	Output Modes .....	72
A.11.2.4	Control Modes.....	72
A.11.2.5	Local Modes .....	72
A.11.2.6	Special Control Characters .....	73
A.12	Utility Conventions.....	73
A.12.1	Utility Argument Syntax.....	73
A.12.2	Utility Syntax Guidelines .....	74
A.13	Headers .....	76
A.13.1	Format of Entries.....	76

<b>Part</b>	<b>B</b>	<b>System Interfaces.....</b>	<b>77</b>
<b>Appendix</b>	<b>B</b>	<b>Rationale for System Interfaces.....</b>	<b>79</b>
	B.1	Introduction.....	79
	B.1.1	Scope.....	79
	B.1.2	Conformance.....	79
	B.1.3	Normative References.....	79
	B.1.4	Change History.....	79
	B.1.5	Terminology.....	85
	B.1.6	Definitions.....	85
	B.1.7	Relationship to Other Formal Standards.....	85
	B.1.8	Portability.....	85
	B.1.8.1	Codes.....	85
	B.1.9	Format of Entries.....	85
	B.2	General Information.....	86
	B.2.1	Use and Implementation of Functions.....	86
	B.2.2	The Compilation Environment.....	87
	B.2.2.1	POSIX.1 Symbols.....	87
	B.2.2.2	The Name Space.....	88
	B.2.3	Error Numbers.....	91
	B.2.3.1	Additional Error Numbers.....	95
	B.2.4	Signal Concepts.....	95
	B.2.4.1	Signal Generation and Delivery.....	97
	B.2.4.2	Realtime Signal Generation and Delivery.....	98
	B.2.4.3	Signal Actions.....	101
	B.2.4.4	Signal Effects on Other Functions.....	104
	B.2.5	Standard I/O Streams.....	105
	B.2.5.1	Interaction of File Descriptors and Standard I/O Streams.....	105
	B.2.5.2	Stream Orientation and Encoding Rules.....	105
	B.2.6	STREAMS.....	105
	B.2.6.1	Accessing STREAMS.....	105
	B.2.7	XSI Interprocess Communication.....	105
	B.2.7.1	IPC General Information.....	106
	B.2.8	Realtime.....	106
	B.2.8.1	Realtime Signals.....	112
	B.2.8.2	Asynchronous I/O.....	114
	B.2.8.3	Memory Management.....	116
	B.2.8.4	Process Scheduling.....	129
	B.2.8.5	Clocks and Timers.....	136
	B.2.9	Threads.....	152
	B.2.9.1	Thread-Safety.....	165
	B.2.9.2	Thread IDs.....	168
	B.2.9.3	Thread Mutexes.....	169
	B.2.9.4	Thread Scheduling.....	169
	B.2.9.5	Thread Cancellation.....	173
	B.2.9.6	Thread Read-Write Locks.....	177
	B.2.9.7	Thread Interactions with Regular File Operations.....	179
	B.2.9.8	Use of Application-Managed Thread Stacks.....	179

B.2.10	Sockets.....	179
B.2.10.1	Address Families.....	179
B.2.10.2	Addressing .....	179
B.2.10.3	Protocols .....	179
B.2.10.4	Routing.....	179
B.2.10.5	Interfaces.....	179
B.2.10.6	Socket Types.....	179
B.2.10.7	Socket I/O Mode.....	180
B.2.10.8	Socket Owner.....	180
B.2.10.9	Socket Queue Limits .....	180
B.2.10.10	Pending Error.....	180
B.2.10.11	Socket Receive Queue.....	180
B.2.10.12	Socket Out-of-Band Data State .....	180
B.2.10.13	Connection Indication Queue .....	180
B.2.10.14	Signals .....	180
B.2.10.15	Asynchronous Errors .....	180
B.2.10.16	Use of Options.....	180
B.2.10.17	Use of Sockets for Local UNIX Connections.....	180
B.2.10.18	Use of Sockets over Internet Protocols.....	180
B.2.10.19	Use of Sockets over Internet Protocols Based on IPv4.....	181
B.2.10.20	Use of Sockets over Internet Protocols Based on IPv6.....	181
B.2.11	Tracing.....	181
B.2.11.1	Objectives .....	181
B.2.11.2	Trace Model.....	186
B.2.11.3	Trace Programming Examples .....	191
B.2.11.4	Rationale on Trace for Debugging.....	199
B.2.11.5	Rationale on Trace Event Type Name Space.....	199
B.2.11.6	Rationale on Trace Events Type Filtering .....	201
B.2.11.7	Tracing, pthread API.....	203
B.2.11.8	Rationale on Triggering .....	204
B.2.11.9	Rationale on Timestamp Clock.....	204
B.2.11.10	Rationale on Different Overrun Conditions.....	205
B.2.12	Data Types.....	205
B.3	System Interfaces .....	208
B.3.1	Examples for Spawn.....	208
<b>Part C</b>	<b>Shell and Utilities.....</b>	<b>219</b>
<b>Appendix C</b>	<b>Rationale for Shell and Utilities .....</b>	<b>221</b>
C.1	Introduction .....	221
C.1.1	Scope.....	221
C.1.2	Conformance .....	221
C.1.3	Normative References .....	221
C.1.4	Change History .....	221
C.1.5	Terminology .....	222
C.1.6	Definitions.....	222
C.1.7	Relationship to Other Documents.....	222
C.1.7.1	System Interfaces.....	222

C.1.7.2	Concepts Derived from the ISO C Standard.....	223
C.1.8	Portability .....	224
C.1.8.1	Codes .....	224
C.1.9	Utility Limits .....	224
C.1.10	Grammar Conventions .....	227
C.1.11	Utility Description Defaults.....	227
C.1.12	Considerations for Utilities in Support of Files of Arbitrary Size ..	231
C.1.13	Built-In Utilities .....	231
C.2	Shell Command Language.....	233
C.2.1	Shell Introduction .....	233
C.2.2	Quoting .....	233
C.2.2.1	Escape Character (Backslash) .....	233
C.2.2.2	Single-Quotes .....	233
C.2.2.3	Double-Quotes .....	233
C.2.3	Token Recognition .....	235
C.2.3.1	Alias Substitution .....	235
C.2.4	Reserved Words .....	236
C.2.5	Parameters and Variables .....	236
C.2.5.1	Positional Parameters.....	236
C.2.5.2	Special Parameters.....	236
C.2.5.3	Shell Variables.....	237
C.2.6	Word Expansions .....	239
C.2.6.1	Tilde Expansion.....	239
C.2.6.2	Parameter Expansion .....	240
C.2.6.3	Command Substitution .....	241
C.2.6.4	Arithmetic Expansion .....	242
C.2.6.5	Field Splitting.....	244
C.2.6.6	Pathname Expansion.....	244
C.2.6.7	Quote Removal.....	245
C.2.7	Redirection .....	245
C.2.7.1	Redirecting Input .....	246
C.2.7.2	Redirecting Output.....	246
C.2.7.3	Appending Redirected Output .....	246
C.2.7.4	Here-Document.....	246
C.2.7.5	Duplicating an Input File Descriptor.....	246
C.2.7.6	Duplicating an Output File Descriptor .....	246
C.2.7.7	Open File Descriptors for Reading and Writing.....	246
C.2.8	Exit Status and Errors .....	246
C.2.8.1	Consequences of Shell Errors .....	246
C.2.8.2	Exit Status for Commands .....	247
C.2.9	Shell Commands .....	247
C.2.9.1	Simple Commands .....	248
C.2.9.2	Pipelines.....	250
C.2.9.3	Lists.....	250
C.2.9.4	Compound Commands.....	252
C.2.9.5	Function Definition Command.....	253
C.2.10	Shell Grammar.....	254
C.2.10.1	Shell Grammar Lexical Conventions.....	255

C.2.10.2	Shell Grammar Rules .....	255
C.2.11	Signals and Error Handling .....	256
C.2.12	Shell Execution Environment .....	256
C.2.13	Pattern Matching Notation .....	256
C.2.13.1	Patterns Matching a Single Character .....	256
C.2.13.2	Patterns Matching Multiple Characters .....	257
C.2.13.3	Patterns Used for Filename Expansion .....	257
C.2.14	Special Built-In Utilities .....	258
C.3	Batch Environment Services and Utilities .....	258
C.3.1	Batch General Concepts .....	261
C.3.2	Batch Services .....	263
C.3.3	Common Behavior for Batch Environment Utilities .....	264
C.4	Utilities .....	264
<b>Part D</b>	<b>Portability Considerations .....</b>	<b>269</b>
<b>Appendix D</b>	<b>Portability Considerations (Informative) .....</b>	<b>271</b>
D.1	User Requirements .....	271
D.1.1	Configuration Interrogation .....	272
D.1.2	Process Management .....	272
D.1.3	Access to Data .....	272
D.1.4	Access to the Environment .....	272
D.1.5	Access to Determinism and Performance Enhancements .....	272
D.1.6	Operating System-Dependent Profile .....	272
D.1.7	I/O Interaction .....	272
D.1.8	Internationalization Interaction .....	273
D.1.9	C-Language Extensions .....	273
D.1.10	Command Language .....	273
D.1.11	Interactive Facilities .....	273
D.1.12	Accomplish Multiple Tasks Simultaneously .....	273
D.1.13	Complex Data Manipulation .....	273
D.1.14	File Hierarchy Manipulation .....	273
D.1.15	Locale Configuration .....	273
D.1.16	Inter-User Communication .....	274
D.1.17	System Environment .....	274
D.1.18	Printing .....	274
D.1.19	Software Development .....	274
D.2	Portability Capabilities .....	274
D.2.1	Configuration Interrogation .....	275
D.2.2	Process Management .....	275
D.2.3	Access to Data .....	276
D.2.4	Access to the Environment .....	276
D.2.5	Bounded (Realtime) Response .....	277
D.2.6	Operating System-Dependent Profile .....	277
D.2.7	I/O Interaction .....	277
D.2.8	Internationalization Interaction .....	277
D.2.9	C-Language Extensions .....	278
D.2.10	Command Language .....	278

D.2.11	Interactive Facilities.....	278
D.2.12	Accomplish Multiple Tasks Simultaneously .....	279
D.2.13	Complex Data Manipulation.....	279
D.2.14	File Hierarchy Manipulation.....	279
D.2.15	Locale Configuration.....	280
D.2.16	Inter-User Communication.....	280
D.2.17	System Environment.....	280
D.2.18	Printing.....	280
D.2.19	Software Development .....	281
D.2.20	Future Growth.....	281
D.3	Profiling Considerations.....	281
D.3.1	Configuration Options.....	281
D.3.2	Configuration Options (Shell and Utilities) .....	282
D.3.3	Configurable Limits.....	283
D.3.4	Configuration Options (System Interfaces) .....	284
D.3.5	Configurable Limits.....	289
D.3.6	Optional Behavior.....	292
<b>Part E</b>	<b>Subprofiling Considerations .....</b>	<b>293</b>
<b>Appendix E</b>	<b>Subprofiling Considerations (Informative).....</b>	<b>295</b>
E.1	Subprofiling Option Groups.....	295
	<b>Index.....</b>	<b>301</b>
 <b>List of Figures</b>		
B-1	Example of a System with Typed Memory .....	124
B-2	Trace System Overview: for Offline Analysis.....	186
B-3	Trace System Overview: for Online Analysis.....	187
B-4	Trace System Overview: States of a Trace Stream .....	189
B-5	Trace Another Process.....	199
B-6	Trace Name Space Overview: With Third-Party Library .....	200
 <b>List of Tables</b>		
A-1	Historical Practice for Symbolic Links .....	31

# *Foreword*

## Structure of the Standard

This standard was originally developed by the Austin Group, a joint working group of members of the IEEE, members of The Open Group, and members of ISO/IEC Joint Technical Committee 1, as one of the four volumes of IEEE Std 1003.1-2001. The standard was approved by ISO and IEC and published in four parts, correlating to the original volumes.

A mapping of the parts to the volumes is shown below:

ISO/IEC 9945 Part	IEEE Std 1003.1 Volume	Description
9945-1	Base Definitions	Includes general terms, concepts, and interfaces common to all parts of ISO/IEC 9945, including utility conventions and C-language header definitions.
9945-2	System Interfaces	Includes definitions for system service functions and subroutines, language-specific system services for the C programming language, function issues, including portability, error handling, and error recovery.
9945-3	Shell and Utilities	Includes definitions for a standard source code-level interface to command interpretation services (a “shell”) and common utility programs for application programs.
9945-4	Rationale	Includes extended rationale that did not fit well into the rest of the document structure, containing historical information concerning the contents of ISO/IEC 9945 and why features were included or discarded by the standard developers.

All four parts comprise the entire standard, and are intended to be used together to accommodate significant internal referencing among them. POSIX-conforming systems are required to support all four parts.

# *Introduction*

**Note:** This introduction is not part of IEEE Std 1003.1-2001, Standard for Information Technology — Portable Operating System Interface (POSIX).

This standard has been jointly developed by the IEEE and The Open Group. It is simultaneously an IEEE Standard, an ISO/IEC Standard, and an Open Group Technical Standard.

## **The Austin Group**

This standard was developed, and is maintained, by a joint working group of members of the IEEE Portable Applications Standards Committee, members of The Open Group, and members of ISO/IEC Joint Technical Committee 1. This joint working group is known as the Austin Group.<sup>3</sup> The Austin Group arose out of discussions amongst the parties which started in early 1998, leading to an initial meeting and formation of the group in September 1998. The purpose of the Austin Group has been to revise, combine, and update the following standards: ISO/IEC 9945-1, ISO/IEC 9945-2, IEEE Std 1003.1, IEEE Std 1003.2, and the Base Specifications of The Open Group Single UNIX Specification.

After two initial meetings, an agreement was signed in July 1999 between The Open Group and the Institute of Electrical and Electronics Engineers (IEEE), Inc., to formalize the project with the first draft of the revised specifications being made available at the same time. Under this agreement, The Open Group and IEEE agreed to share joint copyright of the resulting work. The Open Group has provided the chair and secretariat for the Austin Group.

The base document for the revision was The Open Group's Base volumes of its Single UNIX Specification, Version 2. These were selected since they were a superset of the existing POSIX.1 and POSIX.2 specifications and had some organizational aspects that would benefit the audience for the new revision.

The approach to specification development has been one of “write once, adopt everywhere”, with the deliverables being a set of specifications that carry the IEEE POSIX designation, The Open Group's Technical Standard designation, and an ISO/IEC designation. This set of specifications forms the core of the Single UNIX Specification, Version 3.

This unique development has combined both the industry-led efforts and the formal standardization activities into a single initiative, and included a wide spectrum of participants. The Austin Group continues as the maintenance body for this document.

Anyone wishing to participate in the Austin Group should contact the chair with their request. There are no fees for participation or membership. You may participate as an observer or as a contributor. You do not have to attend face-to-face meetings to participate; electronic participation is most welcome. For more information on the Austin Group and how to participate, see <http://www.opengroup.org/austin>.

---

3. The Austin Group is named after the location of the inaugural meeting held at the IBM facility in Austin, Texas in September 1998.



## Background

The developers of this standard represent a cross section of hardware manufacturers, vendors of operating systems and other software development tools, software designers, consultants, academics, authors, applications programmers, and others.

Conceptually, this standard describes a set of fundamental services needed for the efficient construction of application programs. Access to these services has been provided by defining an interface, using the C programming language, a command interpreter, and common utility programs that establish standard semantics and syntax. Since this interface enables application writers to write portable applications—it was developed with that goal in mind—it has been designated POSIX,<sup>4</sup> an acronym for Portable Operating System Interface.

Although originated to refer to the original IEEE Std 1003.1-1988, the name POSIX more correctly refers to a *family* of related standards: IEEE Std 1003.*n* and the parts of ISO/IEC 9945. In earlier editions of the IEEE standard, the term POSIX was used as a synonym for IEEE Std 1003.1-1988. A preferred term, POSIX.1, emerged. This maintained the advantages of readability of the symbol “POSIX” without being ambiguous with the POSIX family of standards.

## Audience

The intended audience for this standard is all persons concerned with an industry-wide standard operating system based on the UNIX system. This includes at least four groups of people:

1. Persons buying hardware and software systems
2. Persons managing companies that are deciding on future corporate computing directions
3. Persons implementing operating systems, and especially
4. Persons developing applications where portability is an objective

## Purpose

Several principles guided the development of this standard:

- Application-Oriented

The basic goal was to promote portability of application programs across UNIX system environments by developing a clear, consistent, and unambiguous standard for the interface specification of a portable operating system based on the UNIX system documentation. This standard codifies the common, existing definition of the UNIX system.

- Interface, Not Implementation

This standard defines an interface, not an implementation. No distinction is made between library functions and system calls; both are referred to as functions. No details of the implementation of any function are given (although historical practice is sometimes indicated in the RATIONALE section). Symbolic names are given for constants (such as signals and error numbers) rather than numbers.

---

4. The name POSIX was suggested by Richard Stallman. It is expected to be pronounced *pahz-icks*, as in *positive*, not *poh-six*, or other variations. The pronunciation has been published in an attempt to promulgate a standardized way of referring to a standard operating system interface.

- Source, Not Object, Portability

This standard has been written so that a program written and translated for execution on one conforming implementation may also be translated for execution on another conforming implementation. This standard does not guarantee that executable (object or binary) code will execute under a different conforming implementation than that for which it was translated, even if the underlying hardware is identical.

- The C Language

The system interfaces and header definitions are written in terms of the standard C language as specified in the ISO C standard.

- No Superuser, No System Administration

There was no intention to specify all aspects of an operating system. System administration facilities and functions are excluded from this standard, and functions usable only by the superuser have not been included. Still, an implementation of the standard interface may also implement features not in this standard. This standard is also not concerned with hardware constraints or system maintenance.

- Minimal Interface, Minimally Defined

In keeping with the historical design principles of the UNIX system, the mandatory core facilities of this standard have been kept as minimal as possible. Additional capabilities have been added as optional extensions.

- Broadly Implementable

The developers of this standard endeavored to make all specified functions implementable across a wide range of existing and potential systems, including:

1. All of the current major systems that are ultimately derived from the original UNIX system code (Version 7 or later)
2. Compatible systems that are not derived from the original UNIX system code
3. Emulations hosted on entirely different operating systems
4. Networked systems
5. Distributed systems
6. Systems running on a broad range of hardware

No direct references to this goal appear in this standard, but some results of it are mentioned in the Rationale (Informative) volume.

- Minimal Changes to Historical Implementations

When the original version of IEEE Std 1003.1-2001/Cor 2-2004 was published, there were no known historical implementations that did not have to change. However, there was a broad consensus on a set of functions, types, definitions, and concepts that formed an interface that was common to most historical implementations.

The adoption of the 1988 and 1990 IEEE system interface standards, the 1992 IEEE shell and utilities standard, the various Open Group (formerly X/Open) specifications, and the subsequent revisions and addenda to all of them have consolidated this consensus, and this revision reflects the significantly increased level of consensus arrived at since the original versions. The earlier standards and their modifications specified a number of areas where consensus had not been reached before, and these are now reflected in this revision. The authors of the original versions tried, as much as possible, to follow the principles below

when creating new specifications:

1. By standardizing an interface like one in an historical implementation; for example, directories
2. By specifying an interface that is readily implementable in terms of, and backwards-compatible with, historical implementations, such as the extended *tar* format defined in the *pax* utility
3. By specifying an interface that, when added to an historical implementation, will not conflict with it; for example, the *sigaction()* function

This revision tries to minimize the number of changes required to implementations which conform to the earlier versions of the approved standards to bring them into conformance with the current standard. Specifically, the scope of this work excluded doing any “new” work, but rather collecting into a single document what had been spread across a number of documents, and presenting it in what had been proven in practice to be a more effective way. Some changes to prior conforming implementations were unavoidable, primarily as a consequence of resolving conflicts found in prior revisions, or which became apparent when bringing the various pieces together.

However, since it references the 1999 version of the ISO C standard, and no longer supports “Common Usage C”, there are a number of unavoidable changes. Applications portability is similarly affected.

This standard is specifically not a codification of a particular vendor’s product.

It should be noted that implementations will have different kinds of extensions. Some will reflect “historical usage” and will be preserved for execution of pre-existing applications. These functions should be considered “obsolescent” and the standard functions used for new applications. Some extensions will represent functions beyond the scope of this standard. These need to be used with careful management to be able to adapt to future extensions of this standard and/or port to implementations that provide these services in a different manner.

- Minimal Changes to Existing Application Code

A goal of this standard was to minimize additional work for the developers of applications. However, because every known historical implementation will have to change at least slightly to conform, some applications will have to change.

### This Standard

This standard defines the Portable Operating System Interface (POSIX) requirements and consists of the following volumes:

- Base Definitions
- Shell and Utilities
- System Interfaces
- Rationale (Informative) (this volume)

**This Volume**

This volume is being published to assist in the process of review. It contains historical information concerning the contents of this standard and why features were included or discarded by the standard developers. It also contains notes of interest to application programmers on recommended programming practices, emphasizing the consequences of some aspects of this standard that may not be immediately apparent.

This volume is organized in parallel to the normative volumes of this standard, with a separate part for each of the three normative volumes.

Within this volume, the following terms are used:

**base standard**

The portions of this standard that are not optional, equivalent to the definitions of *classic* POSIX.1 and POSIX.2.

**POSIX.0**

Although this term is not used in the normative text of this standard, it is used in this volume to refer to IEEE Std 1003.0-1995.

**POSIX.1b**

Although this term is not used in the normative text of this standard, it is used in this volume to refer to the elements of the POSIX Realtime Extension amendment. (This was earlier referred to as POSIX.4 during the standard development process.)

**POSIX.1c**

Although this term is not used in the normative text of this standard, it is used in this volume to refer to the POSIX Threads Extension amendment. (This was earlier referred to as POSIX.4a during the standard development process.)

**standard developers**

The individuals and companies in the development organizations responsible for this standard: the IEEE P1003.1 working groups, The Open Group Base working group, advised by the hundreds of individual technical experts who balloted the draft standards within the Austin Group, and the member bodies and technical experts of ISO/IEC JTC 1/SC22/WG15.

**XSI extension**

The portions of this standard addressing the extension added for support of the Single UNIX Specification.

# *Participants*

IEEE Std 1003.1-2001 was prepared by the Austin Group, sponsored by the Portable Applications Standards Committee of the IEEE Computer Society, The Open Group, and ISO/SC22 WG15.

## **The Austin Group**

At the time of approval, the membership of the Austin Group was as follows:

**Andrew Josey**, Chair

**Donald W. Cragun**, Organizational Representative, IEEE PASC

**Nicholas Stoughton**, Organizational Representative, ISO/SC22 WG15

**Mark Brown**, Organizational Representative, The Open Group

**Cathy Hughes**, Technical Editor

## **Austin Group Technical Reviewers**

Peter Anvin

Bouazza Bachar

Theodore P. Baker

Walter Briscoe

Mark Brown

Dave Butenhof

Geoff Clare

Donald W. Cragun

Lee Damico

Ulrich Drepper

Paul Eggert

Joanna Farley

Clive D.W. Feather

Andrew Gollan

Michael Gonzalez

Joseph M. Gwinn

Jon Hitchcock

Yvette Ho Sang

Cathy Hughes

Lowell G. Johnson

Andrew Josey

Michael Kavanaugh

David Korn

Marc Aurele La France

Jim Meyering

Gary Miller

Finnbarr P. Murphy

Joseph S. Myers

Sandra O'Donnell

Frank Prindle

Curtis Royster Jr.

Glen Seeds

Keld Jorn Simonsen

Raja Srinivasan

Nicholas Stoughton

Donn S. Terry

Fred Tydeman

Peter Van Der Veen

James Youngman

Jim Zepeda

Jason Zions

# **Austin Group Working Group Members**

Harold C. Adams	Michael Gonzalez	Sandra O'Donnell
Peter Anvin	Karen D. Gordon	Frank Prindle
Pierre-Jean Arcos	Joseph M. Gwinn	Francois Riche
Jay Ashford	Steven A. Haaser	John D. Riley
Bouazza Bachar	Charles E. Hammons	Andrew K. Roach
Theodore P. Baker	Chris J. Harding	Helmut Roth
Robert Barned	Barry Hedquist	Jaideep Roy
Joel Berman	Vincent E. Henley	Curtis Royster Jr.
David J. Blackwood	Karl Heubaum	Stephen C. Schwarm
Shirley Bockstahler-Brandt	Jon Hitchcock	Glen Seeds
James Bottomley	Yvette Ho Sang	Richard Seibel
Walter Briscoe	Niklas Holsti	David L. Shroads Jr.
Andries Brouwer	Thomas Hosmer	W. Olin Sibert
Mark Brown	Cathy Hughes	Keld Jorn Simonsen
Eric W. Burger	Jim D. Isaak	Curtis Smith
Alan Burns	Lowell G. Johnson	Raja Srinivasan
Andries Brouwer	Michael B. Jones	Nicholas Stoughton
Dave Butenhof	Andrew Josey	Marc J. Teller
Keith Chow	Michael J. Karels	Donn S. Terry
Geoff Clare	Michael Kavanaugh	Fred Tydeman
Donald W. Cragun	David Korn	Mark-Rene Uchida
Lee Damico	Steven Kramer	Scott A. Valcourt
Juan Antonio De La Puente	Thomas M. Kurihara	Peter Van Der Veen
Ming De Zhou	Marc Aurele La France	Michael W. Vannier
Steven J. Dovich	C. Douglass Locke	Eric Vought
Richard P. Draves	Nick Maclaren	Frederick N. Webb
Ulrich Drepper	Roger J. Martin	Paul A.T. Wolfgang
Paul Eggert	Craig H. Meyer	Garrett A. Wollman
Philip H. Enslow	Jim Meyering	James Youngman
Joanna Farley	Gary Miller	Oren Yuen
Clive D.W. Feather	Finnbarr P. Murphy	Janusz Zalewski
Pete Forman	Joseph S. Myers	Jim Zepeda
Mark Funkenhauser	John Napier	Jason Zions
Lois Goldthwaite	Peter E. Obermayer	
Andrew Gollan	James T. Oblinger	

### **The Open Group**

When The Open Group approved the Base Specifications, Issue 6 on 12 September 2001, the membership of The Open Group Base Working Group was as follows:

**Andrew Josey**, Chair

**Finnbarr P. Murphy**, Vice-Chair

**Mark Brown**, Austin Group Liaison

**Cathy Hughes**, Technical Editor

### **Base Working Group Members**

Bouazza Bachar

Mark Brown

Dave Butenhof

Donald W. Cragun

Larry Dwyer

Joanna Farley

Andrew Gollan

Karen D. Gordon

Gary Miller

Finnbarr P. Murphy

Frank Prindle

Andrew K. Roach

Curtis Royster Jr.

Nicholas Stoughton

Kenjiro Tsuji

## IEEE

When the IEEE Standards Board approved IEEE Std 1003.1-2001 on 6 December 2001, the membership of the committees was as follows:

### Portable Applications Standards Committee (PASC)

**Lowell G. Johnson**, Chair  
**Joseph M. Gwinn**, Vice-Chair  
**Jay Ashford**, Functional Chair  
**Andrew Josey**, Functional Chair  
**Curtis Royster Jr.**, Functional Chair  
**Nicholas Stoughton**, Secretary

### Balloting Committee

The following members of the balloting committee voted on IEEE Std 1003.1-2001. Balloters may have voted for approval, disapproval, or abstention:

Harold C. Adams	Steven A. Haaser	Frank Prindle
Pierre-Jean Arcos	Charles E. Hammons	Francois Riche
Jay Ashford	Chris J. Harding	John D. Riley
Theodore P. Baker	Barry Hedquist	Andrew K. Roach
Robert Barned	Vincent E. Henley	Helmut Roth
David J. Blackwood	Karl Heubaum	Jaideep Roy
Shirley Bockstahler-Brandt	Niklas Holsti	Curtis Royster Jr.
James Bottomley	Thomas Hosmer	Stephen C. Schwarm
Mark Brown	Jim D. Isaak	Richard Seibel
Eric W. Burger	Lowell G. Johnson	David L. Shroads Jr.
Alan Burns	Michael B. Jones	W. Olin Sibert
Dave Butenhof	Andrew Josey	Keld Jorn Simonsen
Keith Chow	Michael J. Karels	Nicholas Stoughton
Donald W. Cragun	Steven Kramer	Donn S. Terry
Juan Antonio De La Puente	Thomas M. Kurihara	Mark-Rene Uchida
Ming De Zhou	C. Douglass Locke	Scott A. Valcourt
Steven J. Dovich	Roger J. Martin	Michael W. Vannier
Richard P. Draves	Craig H. Meyer	Frederick N. Webb
Philip H. Enslow	Finnbarr P. Murphy	Paul A.T. Wolfgang
Michael Gonzalez	John Napier	Oren Yuen
Karen D. Gordon	Peter E. Obermayer	Janusz Zalewski
Joseph M. Gwinn	James T. Oblinger	

The following organizational representative voted on this standard:

**Andrew Josey**, X/Open Company Ltd.



### IEEE-SA Standards Board

When the IEEE-SA Standards Board approved IEEE Std 1003.1-2001 on 6 December 2001, it had the following membership:

**Donald N. Heirman**, Chair

**James T. Carlo**, Vice-Chair

**Judith Gorman**, Secretary

Satish K. Aggarwal

Mark D. Bowman

Gary R. Engmann

Harold E. Epstein

H. Landis Floyd

Jay Forster\*

Howard M. Frazier

Ruben D. Garzon

James H. Gurney

Richard J. Holleman

Lowell G. Johnson

Robert J. Kennelly

Joseph L. Koepfinger\*

Peter H. Lips

L. Bruce McClung

Daleep C. Mohla

James W. Moore

Robert F. Munzner

Ronald C. Petersen

Gerald H. Peterson

John B. Posey

Gary S. Robinson

Akio Tojo

Donald W. Zipse

Also included are the following non-voting IEEE-SA Standards Board liaisons:

**Alan Cookson**, NIST Representative

**Donald R. Volzka**, TAB Representative

**Yvette Ho Sang**, **Don Messina**, **Savoula Amanatidis**, IEEE Project Editors

---

\* Member Emeritus

IEEE Std 1003.1-2001/Cor 1-2002 was prepared by the Austin Group, sponsored by the Portable Applications Standards Committee of the IEEE Computer Society, The Open Group, and ISO/IEC JTC 1/SC22/WG15.

### **The Austin Group**

At the time of approval, the membership of the Austin Group was as follows:

**Andrew Josey**, Chair

**Donald W. Cragun**, Organizational Representative, IEEE PASC

**Nicholas Stoughton**, Organizational Representative, ISO/IEC JTC 1/SC22/WG15

**Mark Brown**, Organizational Representative, The Open Group

**Cathy Fox**, Technical Editor

### **Austin Group Technical Reviewers**

Theodore P. Baker

Julian Blake

Andries Brouwer

Mark Brown

Dave Butenhof

Geoff Clare

Donald W. Cragun

Ken Dawson

Ulrich Drepper

Larry Dwyer

Paul Eggert

Joanna Farley

Clive D.W. Feather

Cathy Fox

Mark Funkenhauser

Lois Goldthwaite

Andrew Gollan

Michael Gonzalez

Bruno Haible

Ben Harris

Jon Hitchcock

Andreas Jaeger

Andrew Josey

Jonathan Lennox

Nick Maclaren

Jack McCann

Wilhelm Mueller

Joseph S. Myers

Frank Prindle

Kenneth Raeburn

Tim Robbins

Glen Seeds

Matthew Seitz

Keld Jorn Simonsen

Nicholas Stoughton

Alexander Terekhov

Donn S. Terry

Mike Wilson

Garrett A. Wollman

Mark Ziegast

### Austin Group Working Group Members

Harold C. Adams  
Alejandro Alonso  
Jay Ashford  
Theodore P. Baker  
David J. Blackwood  
Julian Blake  
Mitchell Bonnett  
Andries Brouwer  
Mark Brown  
Eric W. Burger  
Alan Burns  
Dave Butenhof  
Keith Chow  
Geoff Clare  
Luis Cordova  
Donald W. Cragun  
Dragan Cvetkovic  
Lee Damico  
Ken Dawson  
Jeroen Dekkers  
Juan Antonio De La Puente  
Steven J. Dovich  
Ulrich Drepper  
Dr. Sourav Dutta  
Larry Dwyer  
Paul Eggert  
Joanna Farley

Clive D.W. Feather  
Yaacov Fenster  
Cathy Fox  
Mark Funkenhauser  
Lois Goldthwaite  
Andrew Gollan  
Michael Gonzalez  
Karen D. Gordon  
Scott Gudgel  
Joseph M. Gwinn  
Steven A. Haaser  
Bruno Haible  
Charles E. Hammons  
Bryan Harold  
Ben Harris  
Barry Hedquist  
Karl Heubaum  
Jon Hitchcock  
Andreas Jaeger  
Andrew Josey  
Kenneth Lang  
Pi-Cheng Law  
Jonathan Lennox  
Nick Maclaren  
Roger J. Martin  
Jack McCann  
George Miao

Wilhelm Mueller  
Finnbarr P. Murphy  
Joseph S. Myers  
Alexey Neyman  
Charles Ngethe  
Peter Petrov  
Frank Prindle  
Vikram Punj  
Kenneth Raeburn  
Francois Riche  
Tim Robbins  
Curtis Royster Jr.  
Diane Schleicher  
Gil Shultz  
Stephen C. Schwarm  
Glen Seeds  
Matthew Seitz  
Keld Jorn Simonsen  
Doug Stevenson  
Nicholas Stoughton  
Alexander Terekhov  
Donn S. Terry  
Mike Wilson  
Garrett A. Wollman  
Oren Yuen  
Mark Ziegast

### **The Open Group**

When The Open Group approved the Base Specifications, Issue 6, Technical Corrigendum 1 on 7 February 2003, the membership of The Open Group Base Working Group was as follows:

**Andrew Josey**, Chair

**Finnbarr P. Murphy**, Vice-Chair

**Mark Brown**, Austin Group Liaison

**Cathy Fox**, Technical Editor

### **Base Working Group Members**

Mark Brown

Dave Butenhof

Donald W. Cragun

Larry Dwyer

Ulrich Drepper

Joanna Farley

Andrew Gollan

Finnbarr P. Murphy

Frank Prindle

Andrew K. Roach

Curtis Royster Jr.

Nicholas Stoughton

Kenjiro Tsuji

### IEEE

When the IEEE Standards Board approved IEEE Std 1003.1-2001/Cor 1-2002 on 11 December 2002, the membership of the committees was as follows:

#### Portable Applications Standards Committee (PASC)

**Lowell G. Johnson**, Chair

**Joseph M. Gwinn**, Vice-Chair

**Jay Ashford**, Functional Chair

**Andrew Josey**, Functional Chair

**Curtis Royster Jr.**, Functional Chair

**Nicholas Stoughton**, Secretary

#### Balloting Committee

The following members of the balloting committee voted on IEEE Std 1003.1-2001/Cor 1-2002. Balloters may have voted for approval, disapproval, or abstention:

Alejandro Alonso  
Jay Ashford  
David J. Blackwood  
Julian Blake  
Mitchell Bonnett  
Mark Brown  
Dave Butenhof  
Keith Chow  
Luis Cordova  
Donald W. Cragun  
Steven J. Dovich  
Dr. Sourav Dutta  
Yaacov Fenster

Michael Gonzalez  
Scott Gudgel  
Charles E. Hammons  
Bryan Harold  
Barry Hedquist  
Karl Heubaum  
Lowell G. Johnson  
Andrew Josey  
Kenneth Lang  
Pi-Cheng Law  
George Miao  
Roger J. Martin  
Finnbarr P. Murphy

Charles Ngethe  
Peter Petrov  
Frank Prindle  
Vikram Punj  
Francois Riche  
Curtis Royster Jr.  
Diane Schleicher  
Stephen C. Schwarm  
Gil Shultz  
Nicholas Stoughton  
Donn S. Terry  
Oren Yuen  
Juan A. de la Puente

## IEEE-SA Standards Board

When the IEEE-SA Standards Board approved IEEE Std 1003.1-2001/Cor 1-2002 on 11 December 2002, the membership was as follows:

**James T. Carlo**, Chair

**James H. Gurney**, Vice-Chair

**Judith Gorman**, Secretary

Sid Bennett

H. Stephen Berger

Clyde R. Camp

Richard DeBlasio

Harold E. Epstein

Julian Forster\*

Howard M. Frazier

Toshio Fukuda

Arnold M. Greenspan

Raymond Hapeman

Donald M. Heirman

Richard H. Hulett

Lowell G. Johnson

Joseph L. Koepfinger\*

Peter H. Lips

Nader Mehravari

Daleep C. Mohla

William J. Moylan

Malcolm V. Thaden

Geoffrey O. Thompson

Howard L. Wolfman

Don Wright

Also included are the following non-voting IEEE-SA Standards Board liaisons:

**Alan Cookson**, NIST Representative

**Satish K. Aggarwal**, NRC Representative

**Savoula Amanatidis**, IEEE Standards Managing Editor

---

\* Member Emeritus

IEEE Std 1003.1-2001/Cor 2-2004 was prepared by the Austin Group, sponsored by the Portable Applications Standards Committee of the IEEE Computer Society, The Open Group, and ISO/IEC JTC 1/SC22/WG15.

### The Austin Group

At the time of approval, the membership of the Austin Group was as follows:

**Andrew Josey**, Chair

**Donald W. Cragun**, Organizational Representative, IEEE PASC

**Nicholas Stoughton**, Organizational Representative, ISO/IEC JTC 1/SC22/WG15

**Mark Brown**, Organizational Representative, The Open Group

**Cathy Fox**, Technical Editor

### Austin Group Technical Reviewers

Jay Ashford

Julian Blake

Mitchell Bonnett

Andries Brouwer

Mark Brown

Paul Buerger

Dave Butenhof

Keith Chow

Geoff Clare

Donald W. Cragun

Lee Damico

Maulik Dave

Juan A. de la Puente

Guru Dutt Dhingra

Loic Domaine

Ulrich Drepper

Sourav Dutta

Larry Dwyer

Paul Eggert

Joanna Farley

Clive D.W. Feather

Yaacov Fenster

Mark Funkenhauser

Ernesto Garcia

Andrew Gollan

Michael Gonzalez

Jean-Denis Gorin

Matthew Gream

Scott Gudgel

Bruno Haible

Charles Hammons

Barry Hedquist

Jon Hitchcock

Lowell G. Johnson

Andrew Josey

Piotr Karocki

David Leciston

Ryan Madron

Roger J. Martin

George Miao

Rajesh Moorkath

Peter Petrov

Franklin Prindle

Vikram Punj

Eusebio Rufian-Zilbermann

Joerg Schilling

Stephen Schwarm

Gil Shultz

Keld Simonsen

Nicholas Stoughton

Alexander Terekhov

Donn Terry

Mark-Rene Uchida

Thomas Unsicker

Scott Valcourt

Mats Wichmann

Garrett A. Wollman

Oren Yuen

Mark Ziegast

**Austin Group Working Group Members**

Harold C. Adams	Mark Funkenhauser	George Miao
Jay Ashford	Ernesto Garcia	Rajesh Moorkath
Theodore P. Baker	Lois Goldthwaite	Vilhelm Mueller
David J. Blackwood	Andrew Gollan	Joseph S. Myers
Julian Blake	Michael Gonzalez	Peter Petrov
Mitchell Bonnett	Karen D. Gordon	Franklin Prindle
Andries Brouwer	Jean-Denis Gorin	Vikram Punj
Mark Brown	Matthew Gream	Kenneth Raeburn
Paul Buerger	Scott Gudgel	Tim Robbins
Alan Burns	Joseph M. Gwinn	Curtis Royster Jr.
Dave Butenhof	Steven A. Haaser	Eusebio Rufian-Zilbermann
Keith Chow	Charles Hammons	Joerg Schilling
Geoff Clare	Ben Harris	Stephen Schwarm
Donald W. Cragun	Barry Hedquist	Glen Seeds
Dragan Cvetkovic	Karl Heubaum	Gil Shultz
Lee Damico	Jon Hitchcock	Keld Simonsen
Maulik Dave	Andreas Jaeger	Nicholas Stoughton
Juan A. de la Puente	Lowell G. Johnson	Alexander Terekhov
Loic Demaille	Andrew Josey	Donn Terry
Steven J. Dovich	Piotr Karocki	Mark-Rene Uchida
Ulrich Drepper	Kenneth Lang	Thomas Unsicker
Guru Dutt Dhingra	Pi-Cheng Law	Scott Valcourt
Sourav Dutta	David Leciston	Mats Wichmann
Larry Dwyer	Wojtek Lerch	Mike Wilson
Paul Eggert	Jonathan Lennox	Garrett A. Wollman
Joanna Farley	Nick Maclaren	Oren Yuen
Clive D.W. Feather	Ryan Madron	Mark Ziegast
Yaacov Fenster	Roger J. Martin	Jason Zions



### The Open Group

When The Open Group approved the Base Specifications, Issue 6, Technical Corrigendum 2 on 18 December 2003, the membership of The Open Group Base Working Group was as follows:

**Andrew Josey**, Chair

**Mark Brown**, Austin Group Liaison

**Cathy Fox**, Technical Editor

### Base Working Group Members

Mark Brown	IBM Corporation
Dave Butenhof	Hewlett-Packard Company
Donald W. Cragun	Sun Microsystems, Inc.
Larry Dwyer	Hewlett-Packard Company
Ulrich Drepper	Red Hat, Inc.
Joanna Farley	Sun Microsystems, Inc.
Andrew Gollan	Sun Microsystems, Inc.
Andrew K. Roach	Sun Microsystems, Inc.
Curtis Royster Jr.	US DoD DISA
Nicholas Stoughton	USENIX Association
Kenjiro Tsuji	Sun Microsystems, Inc.

## IEEE

When the IEEE Standards Board approved IEEE Std 1003.1-2001/Cor 2-2004 on 9 February 2004, the membership of the committees was as follows:

### Portable Applications Standards Committee (PASC)

**Lowell G. Johnson**, Chair

**Joseph M. Gwinn**, Vice-Chair

**Jay Ashford**, Functional Chair

**Andrew Josey**, Functional Chair

**Curtis Royster Jr.**, Functional Chair

**Nicholas Stoughton**, Secretary

### Balloting Committee

The following members of the balloting committee voted on IEEE Std 1003.1-2001/Cor 2-2004. Balloters may have voted for approval, disapproval, or abstention:

Jay Ashford

Julian Blake

Mark Brown

Keith Chow

Donald W. Cragun

Juan A. de la Puente

Guru Dutt Dhingra

Ernesto Garcia

Michael Gonzalez

Jean-Denis Gorin

Matthew Gream

Scott Gudgel

Charles Hammons

Barry Hedquist

Andrew Josey

Piotr Karocki

David Leciston

Ryan Madron

Roger J. Martin

George Miao

Rajesh Moorkath

Peter Petrov

Vikram Punj

Eusebio Rufian-Zilbermann

Stephen Schwarm

Gil Shultz

Mark-Rene Uchida

Thomas Unsicker

Scott Valcourt

Oren Yuen

### IEEE-SA Standards Board

When the IEEE-SA Standards Board approved IEEE Std 1003.1-2001/Cor 2-2004 on 9 February 2004, the membership was as follows:

**Don Wright**, Chair

**Judith Gorman**, Secretary

Chuck Adams

H. Stephen Berger

Mark D. Bowman

Joseph A. Bruder

Bob Davis

Roberto de Boisson

Julian Forster\*

Arnold M. Greenspan

Mark S. Halpin

Raymond Hapeman

Richard J. Holleman

Richard H. Hulett

Lowell G. Johnson

Hermann Koch

Joseph L. Koepfinger\*

Thomas J. McGean

Steve M. Mills

Daleep C. Mohla

Paul Nikolich

T. W. Olsen

Ronald C. Petersen

Gary S. Robinson

Frank Stone

Malcolm V. Thaden

Doug Topping

Joe D. Watson

Also included are the following non-voting IEEE-SA Standards Board liaisons:

**Satish K. Aggarwal**, NRC Representative

**Richard DeBlasio**, DOE Representative

**Alan Cookson**, NIST Representative

**Savoula Amanatidis**, IEEE Standards Managing Editor

---

\* Member Emeritus

# *Trademarks*

The following information is given for the convenience of users of this standard and does not constitute endorsement of these products by The Open Group or the IEEE. There may be other products mentioned in the text that might be covered by trademark protection and readers are advised to verify them independently.

1003.1<sup>™</sup> is a trademark of the Institute of Electrical and Electronic Engineers, Inc.

AIX<sup>®</sup> is a registered trademark of IBM Corporation.

AT&T<sup>®</sup> is a registered trademark of AT&T in the U.S.A. and other countries.

BSD<sup>™</sup> is a trademark of the University of California, Berkeley, U.S.A.

Hewlett-Packard<sup>®</sup>, HP<sup>®</sup>, and HP-UX<sup>®</sup> are registered trademarks of Hewlett-Packard Company.

IBM<sup>®</sup> is a registered trademark of International Business Machines Corporation.

Boundaryless Information Flow is a trademark and UNIX and The Open Group are registered trademarks of The Open Group in the United States and other countries.

All other trademarks are the property of their respective owners.

POSIX<sup>®</sup> is a registered trademark of the Institute of Electrical and Electronic Engineers, Inc.

Sun<sup>®</sup> and Sun Microsystems<sup>®</sup> are registered trademarks of Sun Microsystems, Inc.

/usr/group<sup>®</sup> is a registered trademark of UniForum, the International Network of UNIX System Users.

# *Acknowledgements*

The contributions of the following organizations to the development of IEEE Std 1003.1-2001 are gratefully acknowledged:

- AT&T for permission to reproduce portions of its copyrighted System V Interface Definition (SVID) and material from the UNIX System V Release 2.0 documentation.
- The SC22 WG14 Committees.

This standard was prepared by the Austin Group, a joint working group of the IEEE, The Open Group, and ISO SC22 WG15.

# Referenced Documents

## Normative References

Normative references for this standard are defined in the Base Definitions volume.

## Informative References

The following documents are referenced in this standard:

1984 /usr/group Standard

/usr/group Standards Committee, Santa Clara, CA, UniForum 1984.

Almasi and Gottlieb

George S. Almasi and Allan Gottlieb, *Highly Parallel Computing*, The Benjamin/Cummings Publishing Company, Inc., 1989, ISBN: 0-8053-0177-1.

ANSI C

American National Standard for Information Systems: Standard X3.159-1989, Programming Language C.

ANSI X3.226-1994

American National Standard for Information Systems: Standard X3.226-1994, Programming Language Common LISP.

Brawer

Steven Brawer, *Introduction to Parallel Programming*, Academic Press, 1989, ISBN: 0-12-128470-0.

DeRemer and Pennello Article

DeRemer, Frank and Pennello, Thomas J., *Efficient Computation of LALR(1) Look-Ahead Sets*, SigPlan Notices, Volume 15, No. 8, August 1979.

Draft ANSI X3J11.1

IEEE Floating Point draft report of ANSI X3J11.1 (NCEG).

FIPS 151-1

Federal Information Procurement Standard (FIPS) 151-1. Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language].

FIPS 151-2

Federal Information Procurement Standards (FIPS) 151-2, Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language].

HP-UX Manual

Hewlett-Packard HP-UX Release 9.0 Reference Manual, Third Edition, August 1992.

IEC 60559: 1989

IEC 60559: 1989, Binary Floating-Point Arithmetic for Microprocessor Systems (previously designated IEC 559: 1989).

IEEE Std 754-1985

IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic.

IEEE Std 854-1987

IEEE Std 854-1987, IEEE Standard for Radix-Independent Floating-Point Arithmetic.

## Referenced Documents

### IEEE Std 1003.9-1992

IEEE Std 1003.9-1992, IEEE Standard for Information Technology — POSIX FORTRAN 77 Language Interfaces — Part 1: Binding for System Application Program Interface API.

### IETF RFC 791

Internet Protocol, Version 4 (IPv4), September 1981.

### IETF RFC 819

The Domain Naming Convention for Internet User Applications, Z. Su, J. Postel, August 1982.

### IETF RFC 822

Standard for the Format of ARPA Internet Text Messages, D.H. Crocker, August 1982.

### IETF RFC 919

Broadcasting Internet Datagrams, J. Mogul, October 1984.

### IETF RFC 920

Domain Requirements, J. Postel, J. Reynolds, October 1984.

### IETF RFC 921

Domain Name System Implementation Schedule, J. Postel, October 1984.

### IETF RFC 922

Broadcasting Internet Datagrams in the Presence of Subnets, J. Mogul, October 1984.

### IETF RFC 1034

Domain Names — Concepts and Facilities, P. Mockapetris, November 1987.

### IETF RFC 1035

Domain Names — Implementation and Specification, P. Mockapetris, November 1987.

### IETF RFC 1123

Requirements for Internet Hosts — Application and Support, R. Braden, October 1989.

### IETF RFC 1886

DNS Extensions to Support Internet Protocol, Version 6 (IPv6), C. Huitema, S. Thomson, December 1995.

### IETF RFC 2045

Multipurpose Internet Mail Extensions (MIME), Part 1: Format of Internet Message Bodies, N. Freed, N. Borenstein, November 1996.

### IETF RFC 2181

Clarifications to the DNS Specification, R. Elz, R. Bush, July 1997.

### IETF RFC 2373

Internet Protocol, Version 6 (IPv6) Addressing Architecture, S. Deering, R. Hinden, July 1998.

### IETF RFC 2460

Internet Protocol, Version 6 (IPv6), S. Deering, R. Hinden, December 1998.

### Internationalisation Guide

Guide, July 1993, Internationalisation Guide, Version 2 (ISBN: 1-859120-02-4, G304), published by The Open Group.

### ISO C (1990)

ISO/IEC 9899:1990, Programming Languages — C, including Amendment 1:1995 (E), C Integrity (Multibyte Support Extensions (MSE) for ISO C).

ISO 2375:1985

ISO 2375:1985, Data Processing — Procedure for Registration of Escape Sequences.

ISO 8652:1987

ISO 8652:1987, Programming Languages — Ada (technically identical to ANSI standard 1815A-1983).

ISO/IEC 1539:1990

ISO/IEC 1539:1990, Information Technology — Programming Languages — Fortran (technically identical to the ANSI X3.9-1978 standard [FORTRAN 77]).

ISO/IEC 4873:1991

ISO/IEC 4873:1991, Information Technology — ISO 8-bit Code for Information Interchange — Structure and Rules for Implementation.

ISO/IEC 6429:1992

ISO/IEC 6429:1992, Information Technology — Control Functions for Coded Character Sets.

ISO/IEC 6937:1994

ISO/IEC 6937:1994, Information Technology — Coded Character Set for Text Communication — Latin Alphabet.

ISO/IEC 8802-3:1996

ISO/IEC 8802-3:1996, Information Technology — Telecommunications and Information Exchange Between Systems — Local and Metropolitan Area Networks — Specific Requirements — Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications.

ISO/IEC 8859

ISO/IEC 8859, Information Technology — 8-Bit Single-Byte Coded Graphic Character Sets:

Part 1: Latin Alphabet No. 1

Part 2: Latin Alphabet No. 2

Part 3: Latin Alphabet No. 3

Part 4: Latin Alphabet No. 4

Part 5: Latin/Cyrillic Alphabet

Part 6: Latin/Arabic Alphabet

Part 7: Latin/Greek Alphabet

Part 8: Latin/Hebrew Alphabet

Part 9: Latin Alphabet No. 5

Part 10: Latin Alphabet No. 6

Part 11: Latin/Thai Alphabet

Part 13: Latin Alphabet No. 7

Part 14: Latin Alphabet No. 8

Part 15: Latin Alphabet No. 9

Part 16: Latin Alphabet No. 10

ISO POSIX-1:1996

ISO/IEC 9945-1:1996, Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language] (identical to ANSI/IEEE Std 1003.1-1996). Incorporating ANSI/IEEE Stds 1003.1-1990, 1003.1b-1993, 1003.1c-1995, and 1003.1i-1995.

ISO POSIX-2:1993

ISO/IEC 9945-2:1993, Information Technology — Portable Operating System Interface (POSIX) — Part 2: Shell and Utilities (identical to ANSI/IEEE Std 1003.2-1992, as amended



by ANSI/IEEE Std 1003.2a-1992).

### Issue 1

X/Open Portability Guide, July 1985 (ISBN: 0-444-87839-4).

### Issue 2

X/Open Portability Guide, January 1987:

- Volume 1: XVS Commands and Utilities (ISBN: 0-444-70174-5)
- Volume 2: XVS System Calls and Libraries (ISBN: 0-444-70175-3)

### Issue 3

X/Open Specification, 1988, 1989, February 1992:

- Commands and Utilities, Issue 3 (ISBN: 1-872630-36-7, C211); this specification was formerly X/Open Portability Guide, Issue 3, Volume 1, January 1989, XSI Commands and Utilities (ISBN: 0-13-685835-X, XO/XPG/89/002)
- System Interfaces and Headers, Issue 3 (ISBN: 1-872630-37-5, C212); this specification was formerly X/Open Portability Guide, Issue 3, Volume 2, January 1989, XSI System Interface and Headers (ISBN: 0-13-685843-0, XO/XPG/89/003)
- Curses Interface, Issue 3, contained in Supplementary Definitions, Issue 3 (ISBN: 1-872630-38-3, C213), Chapters 9 to 14 inclusive; this specification was formerly X/Open Portability Guide, Issue 3, Volume 3, January 1989, XSI Supplementary Definitions (ISBN: 0-13-685850-3, XO/XPG/89/004)
- Headers Interface, Issue 3, contained in Supplementary Definitions, Issue 3 (ISBN: 1-872630-38-3, C213), Chapter 19, Cpio and Tar Headers; this specification was formerly X/Open Portability Guide Issue 3, Volume 3, January 1989, XSI Supplementary Definitions (ISBN: 0-13-685850-3, XO/XPG/89/004)

### Issue 4

CAE Specification, July 1992, published by The Open Group:

- System Interface Definitions (XBD), Issue 4 (ISBN: 1-872630-46-4, C204)
- Commands and Utilities (XCU), Issue 4 (ISBN: 1-872630-48-0, C203)
- System Interfaces and Headers (XSH), Issue 4 (ISBN: 1-872630-47-2, C202)

### Issue 4, Version 2

CAE Specification, August 1994, published by The Open Group:

- System Interface Definitions (XBD), Issue 4, Version 2 (ISBN: 1-85912-036-9, C434)
- Commands and Utilities (XCU), Issue 4, Version 2 (ISBN: 1-85912-034-2, C436)
- System Interfaces and Headers (XSH), Issue 4, Version 2 (ISBN: 1-85912-037-7, C435)

### Issue 5

Technical Standard, February 1997, published by The Open Group:

- System Interface Definitions (XBD), Issue 5 (ISBN: 1-85912-186-1, C605)
- Commands and Utilities (XCU), Issue 5 (ISBN: 1-85912-191-8, C604)
- System Interfaces and Headers (XSH), Issue 5 (ISBN: 1-85912-181-0, C606)

### Knuth Article

Knuth, Donald E., *On the Translation of Languages from Left to Right*, Information and Control, Volume 8, No. 6, October 1965.

KornShell

Bolsky, Morris I. and Korn, David G., *The New KornShell Command and Programming Language*, March 1995, Prentice Hall.

MSE Working Draft

Working draft of ISO/IEC 9899:1990/Add3:Draft, Addendum 3 — Multibyte Support Extensions (MSE) as documented in the ISO Working Paper SC22/WG14/N205 dated 31 March 1992.

POSIX.0: 1995

IEEE Std 1003.0-1995, IEEE Guide to the POSIX Open System Environment (OSE) (identical to ISO/IEC TR 14252).

POSIX.1: 1988

IEEE Std 1003.1-1988, IEEE Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language].

POSIX.1: 1990

IEEE Std 1003.1-1990, IEEE Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language].

POSIX.1a

P1003.1a, Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) — (C Language) Amendment.

POSIX.1d: 1999

IEEE Std 1003.1d-1999, IEEE Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) — Amendment 4: Additional Realtime Extensions [C Language].

POSIX.1g: 2000

IEEE Std 1003.1g-2000, IEEE Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) — Amendment 6: Protocol-Independent Interfaces (PII).

POSIX.1j: 2000

IEEE Std 1003.1j-2000, IEEE Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) — Amendment 5: Advanced Realtime Extensions [C Language].

POSIX.1q: 2000

IEEE Std 1003.1q-2000, IEEE Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) — Amendment 7: Tracing [C Language].

POSIX.2b

P1003.2b, Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 2: Shell and Utilities — Amendment.

POSIX.2d:-1994

IEEE Std 1003.2d-1994, IEEE Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 2: Shell and Utilities — Amendment 1: Batch Environment.

## Referenced Documents

### POSIX.13:-1998

IEEE Std 1003.13:1998, IEEE Standard for Information Technology — Standardized Application Environment Profile (AEP) — POSIX Realtime Application Support.

### Sarwate Article

Sarwate, Dilip V., *Computation of Cyclic Redundancy Checks via Table Lookup*, Communications of the ACM, Volume 30, No. 8, August 1988.

### Sprunt, Sha, and Lehoczky

Sprunt, B., Sha, L., and Lehoczky, J.P., *Aperiodic Task Scheduling for Hard Real-Time Systems*, The Journal of Real-Time Systems, Volume 1, 1989, Pages 27-60.

### SVID, Issue 1

American Telephone and Telegraph Company, System V Interface Definition (SVID), Issue 1; Morristown, NJ, UNIX Press, 1985.

### SVID, Issue 2

American Telephone and Telegraph Company, System V Interface Definition (SVID), Issue 2; Morristown, NJ, UNIX Press, 1986.

### SVID, Issue 3

American Telephone and Telegraph Company, System V Interface Definition (SVID), Issue 3; Morristown, NJ, UNIX Press, 1989.

### The AWK Programming Language

Aho, Alfred V., Kernighan, Brian W., and Weinberger, Peter J., *The AWK Programming Language*, Reading, MA, Addison-Wesley 1988.

### UNIX Programmer's Manual

American Telephone and Telegraph Company, *UNIX Time-Sharing System: UNIX Programmer's Manual*, 7th Edition, Murray Hill, NJ, Bell Telephone Laboratories, January 1979.

### XNS, Issue 4

CAE Specification, August 1994, Networking Services, Issue 4 (ISBN: 1-85912-049-0, C438), published by The Open Group.

### XNS, Issue 5

CAE Specification, February 1997, Networking Services, Issue 5 (ISBN: 1-85912-165-9, C523), published by The Open Group.

### XNS, Issue 5.2

Technical Standard, January 2000, Networking Services (XNS), Issue 5.2 (ISBN: 1-85912-241-8, C808), published by The Open Group.

### X/Open Curses, Issue 4, Version 2

CAE Specification, May 1996, X/Open Curses, Issue 4, Version 2 (ISBN: 1-85912-171-3, C610), published by The Open Group.

### Yacc

*Yacc: Yet Another Compiler Compiler*, Stephen C. Johnson, 1978.

## **Source Documents**

Parts of the following documents were used to create the base documents for this standard:

### **AIX 3.2 Manual**

AIX Version 3.2 For RISC System/6000, Technical Reference: Base Operating System and Extensions, 1990, 1992 (Part No. SC23-2382-00).

### **OSF/1**

OSF/1 Programmer's Reference, Release 1.2 (ISBN: 0-13-020579-6).

### **OSF AES**

Application Environment Specification (AES) Operating System Programming Interfaces Volume, Revision A (ISBN: 0-13-043522-8).

### **System V Release 2.0**

- UNIX System V Release 2.0 Programmer's Reference Manual (April 1984 - Issue 2).
- UNIX System V Release 2.0 Programming Guide (April 1984 - Issue 2).

### **System V Release 4.2**

Operating System API Reference, UNIX SVR4.2 (1992) (ISBN: 0-13-017658-3).

# *Rationale (Informative)*

## **Part A:**

### **Base Definitions**

*The Open Group*

*The Institute of Electrical and Electronics Engineers, Inc.*



# Rationale for Base Definitions

## A.1 Introduction

### A.1.1 Scope

IEEE Std 1003.1-2001 is one of a family of standards known as POSIX. The family of standards extends to many topics; IEEE Std 1003.1-2001 is known as POSIX.1 and consists of both operating system interfaces and shell and utilities. IEEE Std 1003.1-2001 is technically identical to The Open Group Base Specifications, Issue 6, which comprise the core volumes of the Single UNIX Specification, Version 3.

#### Scope of IEEE Std 1003.1-2001

The (paraphrased) goals of this development were to produce a single common revision to the overlapping POSIX.1 and POSIX.2 standards, and the Single UNIX Specification, Version 2. As such, the scope of the revision includes the scopes of the original documents merged.

Since the revision includes merging the Base volumes of the Single UNIX Specification, many features that were previously not “adopted” into earlier revisions of POSIX.1 and POSIX.2 are now included in IEEE Std 1003.1-2001. In most cases, these additions are part of the XSI extension; in other cases the standard developers decided that now was the time to migrate these to the base standard.

The Single UNIX Specification programming environment provides a broad-based functional set of interfaces to support the porting of existing UNIX applications and the development of new applications. The environment also supports a rich set of tools for application development.

The majority of the obsolescent material from the existing POSIX.1 and POSIX.2 standards, and material marked LEGACY from The Open Group's Base specifications, has been removed in this revision. New members of the Legacy Option Group have been added, reflecting the advance in understanding of what is required.

The following IEEE standards have been added to the base documents in this revision:

- IEEE Std 1003.1d-1999
- IEEE Std 1003.1j-2000
- IEEE Std 1003.1q-2000
- IEEE P1003.1a draft standard
- IEEE Std 1003.2d-1994
- IEEE P1003.2b draft standard
- Selected parts of IEEE Std 1003.1g-2000

Only selected parts of IEEE Std 1003.1g-2000 were included. This was because there is much duplication between the XNS, Issue 5.2 specification (another base document) and the material from IEEE Std 1003.1g-2000, the former document being aligned with the latest networking specifications for IPv6. Only the following sections of IEEE Std 1003.1g-2000 were considered for inclusion:

- General terms related to sockets (Section 2.2.2)
- Socket concepts (Sections 5.1 through 5.3 inclusive)
- The *pselect()* function (Sections 6.2.2.1 and 6.2.3)
- The `<sys/select.h>` header (Section 6.2)

The following were requirements on IEEE Std 1003.1-2001:

- Backward-compatibility

It was agreed that there should be no breakage of functionality in the existing base documents. This requirement was tempered by changes introduced due to interpretations and corrigenda on the base documents, and any changes introduced in the ISO/IEC 9899:1999 standard (C Language).

- Architecture and n-bit neutral

The common standard should not make any implicit assumptions about the system architecture or size of data types; for example, previously some 32-bit implicit assumptions had crept into the standards.

- Extensibility

It should be possible to extend the common standard without breaking backwards-compatibility. For example, the name space should be reserved and structured to avoid duplication of names between the standard and extensions to it.

## POSIX.1 and the ISO C Standard

Previous revisions of POSIX.1 built upon the ISO C standard by reference only. This revision takes a different approach.

The standard developers believed it essential for a programmer to have a single complete reference place, but recognized that deference to the formal standard had to be addressed for the duplicate interface definitions between the ISO C standard and the Single UNIX Specification.

It was agreed that where an interface has a version in the ISO C standard, the DESCRIPTION section should describe the relationship to the ISO C standard and markings should be added as appropriate to show where the ISO C standard has been extended in the text.

A block of text was added to the start of each affected reference page stating whether the page is aligned with the ISO C standard or extended. Each page was parsed for additions beyond the ISO C standard (that is, including both POSIX and UNIX extensions), and these extensions are marked as CX extensions (for C Extensions).

## FIPS Requirements

The Federal Information Processing Standards (FIPS) are a series of U.S. government procurement standards managed and maintained on behalf of the U.S. Department of Commerce by the National Institute of Standards and Technology (NIST).

The following restrictions have been made in this version of IEEE Std 1003.1 in order to align with FIPS 151-2 requirements:

- The implementation supports `_POSIX_CHOWN_RESTRICTED`.
- The limit `{NGROUPS_MAX}` is now greater than or equal to 8.
- The implementation supports the setting of the group ID of a file (when it is created) to that of the parent directory.



- The implementation supports `_POSIX_SAVED_IDS`.
- The implementation supports `_POSIX_VDISABLE`.
- The implementation supports `_POSIX_JOB_CONTROL`.
- The implementation supports `_POSIX_NO_TRUNC`.
- The `read()` function returns the number of bytes read when interrupted by a signal and does not return `-1`.
- The `write()` function returns the number of bytes written when interrupted by a signal and does not return `-1`.
- In the environment for the login shell, the environment variables `LOGNAME` and `HOME` are defined and have the properties described in IEEE Std 1003.1-2001.
- The value of `{CHILD_MAX}` is now greater than or equal to 25.
- The value of `{OPEN_MAX}` is now greater than or equal to 20.
- The implementation supports the functionality associated with the symbols `CS7`, `CS8`, `CSTOPB`, `PARODD`, and `PARENB` defined in `<termios.h>`.

### A.1.2 Conformance

See Section A.2 (on page 9).

### A.1.3 Normative References

There is no additional rationale provided for this section.

### A.1.4 Terminology

The meanings specified in IEEE Std 1003.1-2001 for the words *shall*, *should*, and *may* are mandated by ISO/IEC directives.

In the Rationale (Informative) volume of IEEE Std 1003.1-2001, the words *shall*, *should*, and *may* are sometimes used to illustrate similar usages in IEEE Std 1003.1-2001. However, the rationale itself does not specify anything regarding implementations or applications.

#### conformance document

As a practical matter, the conformance document is effectively part of the system documentation. Conformance documents are distinguished by IEEE Std 1003.1-2001 so that they can be referred to distinctly.

#### implementation-defined

This definition is analogous to that of the ISO C standard and, together with “undefined” and “unspecified”, provides a range of specification of freedom allowed to the interface implementor.

#### may

The use of *may* has been limited as much as possible, due both to confusion stemming from its ordinary English meaning and to objections regarding the desirability of having as few options as possible and those as clearly specified as possible.

The usage of *can* and *may* were selected to contrast optional application behavior (can) against optional implementation behavior (may).

**shall**

Declarative sentences are sometimes used in IEEE Std 1003.1-2001 as if they included the word *shall*, and facilities thus specified are no less required. For example, the two statements:

1. The *foo()* function shall return zero.
2. The *foo()* function returns zero.

are meant to be exactly equivalent.

**should**

In IEEE Std 1003.1-2001, the word *should* does not usually apply to the implementation, but rather to the application. Thus, the important words regarding implementations are *shall*, which indicates requirements, and *may*, which indicates options.

**obsolescent**

The term “obsolescent” means “do not use this feature in new applications”. The obsolescence concept is not an ideal solution, but was used as a method of increasing consensus: many more objections would be heard from the user community if some of these historical features were suddenly withdrawn without the grace period obsolescence implies. The phrase “may be considered for withdrawal in future revisions” implies that the result of that consideration might in fact keep those features indefinitely if the predominance of applications do not migrate away from them quickly.

**legacy**

The term “legacy” was added for compatibility with the Single UNIX Specification. It means “this feature is historic and optional; do not use this feature in new applications. There are alternative interfaces that are more suitable.”. It is used exclusively for XSI extensions, and includes facilities that were mandatory in previous versions of the base document but are optional in this revision. This is a way to “sunset” the usage of certain functions. Application writers should not rely on the existence of these facilities in new applications, but should follow the migration path detailed in the APPLICATION USAGE sections of the relevant pages.

The terms “legacy” and “obsolescent” are different: a feature marked LEGACY is not recommended for new work and need not be present on an implementation (if the XSI Legacy Option Group is not supported). A feature noted as obsolescent is supported by all implementations, but may be removed in a future revision; new applications should not use these features.

**system documentation**

The system documentation should normally describe the whole of the implementation, including any extensions provided by the implementation. Such documents normally contain information at least as detailed as the specifications in IEEE Std 1003.1-2001. Few requirements are made on the system documentation, but the term is needed to avoid a dangling pointer where the conformance document is permitted to point to the system documentation.

**undefined**

See *implementation-defined*.

**unspecified**

See *implementation-defined*.

The definitions for “unspecified” and “undefined” appear nearly identical at first examination, but are not. The term “unspecified” means that a conforming application may deal with the unspecified behavior, and it should not care what the outcome is. The term

“undefined” says that a conforming application should not do it because no definition is provided for what it does (and implicitly it would care what the outcome was if it tried it). It is important to remember, however, that if the syntax permits the statement at all, it must have some outcome in a real implementation.

Thus, the terms “undefined” and “unspecified” apply to the way the application should think about the feature. In terms of the implementation, it is always “defined”—there is always some result, even if it is an error. The implementation is free to choose the behavior it prefers.

This also implies that an implementation, or another standard, could specify or define the result in a useful fashion. The terms apply to IEEE Std 1003.1-2001 specifically.

The term “implementation-defined” implies requirements for documentation that are not required for “undefined” (or “unspecified”). Where there is no need for a conforming program to know the definition, the term “undefined” is used, even though “implementation-defined” could also have been used in this context. There could be a fourth term, specifying “this standard does not say what this does; it is acceptable to define it in an implementation, but it does not need to be documented”, and undefined would then be used very rarely for the few things for which any definition is not useful. In particular, implementation-defined is used where it is believed that certain classes of application will need to know such details to determine whether the application can be successfully ported to the implementation. Such applications are not always strictly portable, but nevertheless are common and useful; often the requirements met by the application cannot be met without dealing with the issues implied by “implementation-defined”. In some places the text refers to facilities supplied by the implementation that are outside the standard as implementation-supplied or implementation-provided. This is not intended to imply a requirement for documentation. If it were, the term “implementation-defined” would have been used.

In many places IEEE Std 1003.1-2001 is silent about the behavior of some possible construct. For example, a variable may be defined for a specified range of values and behaviors are described for those values; nothing is said about what happens if the variable has any other value. That kind of silence can imply an error in the standard, but it may also imply that the standard was intentionally silent and that any behavior is permitted. There is a natural tendency to infer that if the standard is silent, a behavior is prohibited. That is not the intent. Silence is intended to be equivalent to the term “unspecified”.

The term “application” is not defined in IEEE Std 1003.1-2001; it is assumed to be a part of general computer science terminology.

Three terms used within IEEE Std 1003.1-2001 overlap in meaning: “macro”, “symbolic name”, and “symbolic constant”.

#### **macro**

This usually describes a C preprocessor symbol, the result of the **#define** operator, with or without an argument. It may also be used to describe similar mechanisms in editors and text processors.

#### **symbolic name**

This can also refer to a C preprocessor symbol (without arguments), but is also used to refer to the names for characters in character sets. In addition, it is sometimes used to refer to host names and even filenames.

#### **symbolic constant**

This also refers to a C preprocessor symbol (also without arguments).

217 In most cases, the difference in semantic content is negligible to nonexistent. Readers should not  
 218 attempt to read any meaning into the various usages of these terms.

## 219 **A.1.5 Portability**

220 To aid the identification of options within IEEE Std 1003.1-2001, a notation consisting of margin  
 221 codes and shading is used. This is based on the notation used in previous revisions of The Open  
 222 Group's Base specifications.

223 The benefit of this approach is a reduction in the number of *if* statements within the running  
 224 text, that makes the text easier to read, and also an identification to the programmer that they  
 225 need to ensure that their target platforms support the underlying options. For example, if  
 226 functionality is marked with THR in the margin, it will be available on all systems supporting  
 227 the Threads option, but may not be available on some others.

### 228 **A.1.5.1 Codes**

229 This section includes codes for options defined in the Base Definitions volume of  
 230 IEEE Std 1003.1-2001, Section 2.1.6, Options, and the following additional codes for other  
 231 purposes:

232 CX This margin code is used to denote extensions beyond the ISO C standard. For  
 233 interfaces that are duplicated between IEEE Std 1003.1-2001 and the ISO C standard, a  
 234 CX introduction block describes the nature of the duplication, with any extensions  
 235 appropriately CX marked and shaded.

236 Where an interface is added to an ISO C standard header, within the header the  
 237 interface has an appropriate margin marker and shading (for example, CX, XSI, TSF,  
 238 and so on) and the same marking appears on the reference page in the SYNOPSIS  
 239 section. This enables a programmer to easily identify that the interface is extending an  
 240 ISO C standard header.

241 MX This margin code is used to denote IEC 60559: 1989 standard floating-point extensions.

242 OB This margin code is used to denote obsolescent behavior and thus flag a possible future  
 243 applications portability warning.

244 OH The Single UNIX Specification has historically tried to reduce the number of headers an  
 245 application has had to include when using a particular interface. Sometimes this was  
 246 fewer than the base standard, and hence a notation is used to flag which headers are  
 247 optional if you are using a system supporting the XSI extension.

248 XSI This code is used to denote interfaces and facilities within interfaces only required on  
 249 systems supporting the XSI extension. This is introduced to support the Single UNIX  
 250 Specification.

251 XSR This code is used to denote interfaces and facilities within interfaces only required on  
 252 systems supporting STREAMS. This is introduced to support the Single UNIX  
 253 Specification, although it is defined in a way so that it can stand alone from the XSI  
 254 notation.

### 255 A.1.5.2 Margin Code Notation

256 Since some features may depend on one or more options, or require more than one option, a  
 257 notation is used. Where a feature requires support of a single option, a single margin code will  
 258 occur in the margin. If it depends on two options and both are required, then the codes will  
 259 appear with a <space> separator. If either of two options are required, then a logical OR is  
 260 denoted using the ' | ' symbol. If more than two codes are used, a special notation is used.

## 261 A.2 Conformance

262 The terms “profile” and “profiling” are used throughout this section.

263 A profile of a standard or standards is a codified set of option selections, such that by being  
 264 conformant to a profile, particular classes of users are specifically supported.

265 These conformance definitions are descended from those in the ISO POSIX-1: 1996 standard, but  
 266 with changes for the following:

- 267 • The addition of profiling options, allowing larger profiles of options such as the XSI  
 268 extension used by the Single UNIX Specification. In effect, it has profiled itself (that is,  
 269 created a self-profile).
- 270 • The addition of a definition of subprofiling considerations, to allow smaller profiles of  
 271 options.
- 272 • The addition of a hierarchy of super-options for XSI; these were formerly known as “Feature  
 273 Groups” in the System Interfaces and Headers, Issue 5 specification.
- 274 • Options from the ISO POSIX-2: 1993 standard are also now included, as IEEE Std 1003.1-2001  
 275 merges the functionality from it.

### 276 A.2.1 Implementation Conformance

277 These definitions allow application developers to know what to depend on in an  
 278 implementation.

279 There is no definition of a “strictly conforming implementation”; that would be an  
 280 implementation that provides *only* those facilities specified by POSIX.1 with no extensions  
 281 whatsoever. This is because no actual operating system implementation can exist without  
 282 system administration and initialization facilities that are beyond the scope of POSIX.1.

#### 283 A.2.1.1 Requirements

284 The word “support” is used in certain instances, rather than “provide”, in order to allow an  
 285 implementation that has no resident software development facilities, but that supports the  
 286 execution of a *Strictly Conforming POSIX.1 Application*, to be a *conforming implementation*.

#### 287 A.2.1.2 Documentation

288 The conformance documentation is required to use the same numbering scheme as POSIX.1 for  
 289 purposes of cross-referencing. All options that an implementation chooses are reflected in  
 290 <limits.h> and <unistd.h>.

291 Note that the use of “may” in terms of where conformance documents record where  
 292 implementations may vary, implies that it is not required to describe those features identified as  
 293 undefined or unspecified.

Other aspects of systems must be evaluated by purchasers for suitability. Many systems incorporate buffering facilities, maintaining updated data in volatile storage and transferring such updates to non-volatile storage asynchronously. Various exception conditions, such as a power failure or a system crash, can cause this data to be lost. The data may be associated with a file that is still open, with one that has been closed, with a directory, or with any other internal system data structures associated with permanent storage. This data can be lost, in whole or part, so that only careful inspection of file contents could determine that an update did not occur.

Also, interrelated file activities, where multiple files and/or directories are updated, or where space is allocated or released in the file system structures, can leave inconsistencies in the relationship between data in the various files and directories, or in the file system itself. Such inconsistencies can break applications that expect updates to occur in a specific sequence, so that updates in one place correspond with related updates in another place.

For example, if a user creates a file, places information in the file, and then records this action in another file, a system or power failure at this point followed by restart may result in a state in which the record of the action is permanently recorded, but the file created (or some of its information) has been lost. The consequences of this to the user may be undesirable. For a user on such a system, the only safe action may be to require the system administrator to have a policy that requires, after any system or power failure, that the entire file system must be restored from the most recent backup copy (causing all intervening work to be lost).

The characteristics of each implementation will vary in this respect and may or may not meet the requirements of a given application or user. Enforcement of such requirements is beyond the scope of POSIX.1. It is up to the purchaser to determine what facilities are provided in an implementation that affect the exposure to possible data or sequence loss, and also what underlying implementation techniques and/or facilities are provided that reduce or limit such loss or its consequences.

### A.2.1.3 *POSIX Conformance*

This really means conformance to the base standard; however, since this revision includes the core material of the Single UNIX Specification, the standard developers decided that it was appropriate to segment the conformance requirements into two, the former for the base standard, and the latter for the Single UNIX Specification.

Within POSIX.1 there are some symbolic constants that, if defined, indicate that a certain option is enabled. Other symbolic constants exist in POSIX.1 for other reasons.

As part of the revision some alignment has occurred of the options with the FIPS 151-2 profile on the POSIX.1-1990 standard. The following options from the POSIX.1-1990 standard are now mandatory:

- `_POSIX_JOB_CONTROL`
- `_POSIX_SAVED_IDS`
- `_POSIX_VDISABLE`

A POSIX-conformant system may support the XSI extensions of the Single UNIX Specification. This was intentional since the standard developers intend them to be upwards-compatible, so that a system conforming to the Single UNIX Specification can also conform to the base standard at the same time.

#### A.2.1.4 XSI Conformance

This section is added since the revision merges in the base volumes of the Single UNIX Specification.

XSI conformance can be thought of as a profile, selecting certain options from IEEE Std 1003.1-2001.

#### A.2.1.5 Option Groups

The concept of “Option Groups” is introduced to IEEE Std 1003.1-2001 to allow collections of related functions or options to be grouped together. This has been used as follows: the “XSI Option Groups” have been created to allow super-options, collections of underlying options and related functions, to be collectively supported by XSI-conforming systems. These reflect the “Feature Groups” from the System Interfaces and Headers, Issue 5 specification.

The standard developers considered the matter of subprofiling and decided it was better to include an enabling mechanism rather than detailed normative requirements. A set of subprofiling options was developed and included later in this volume of IEEE Std 1003.1-2001 as an informative illustration.

#### Subprofiling Considerations

The goal of not simultaneously fixing maximums and minimums was to allow implementations of the base standard or standards to support multiple profiles without conflict.

The following summarizes the rules for the limit types:

Limit Type	Fixed Value	Minimum Acceptable Value	Maximum Acceptable Value
Standard Profile	$X_s$ $X_p == X_s$ (No change)	$Y_s$ $Y_p \geq Y_s$ (May increase the limit)	$Z_s$ $Z_p \leq Z_s$ (May decrease the limit)

The intent is that ranges specified by limits in profiles be entirely contained within the corresponding ranges of the base standard or standards being profiled, and that the unlimited end of a range in a base standard must remain unlimited in any profile of that standard.

Thus, the fixed `_POSIX_*` limits are constants and must not be changed by a profile. The variable counterparts (typically without the leading `_POSIX_`) can be changed but still remain semantically the same; that is, they still allow implementation values to vary as long as they meet the requirements for that value (be it a minimum or maximum).

Where a profile does not provide a feature upon which a limit is based, the limit is not relevant. Applications written to that profile should be written to operate independently of the value of the limit.

An example which has previously allowed implementations to support both the base standard and two other profiles in a compatible manner follows:

```
Base standard (POSIX.1-1996): _POSIX_CHILD_MAX 6
Base standard: CHILD_MAX minimum maximum _POSIX_CHILD_MAX
FIPS profile/SUSv2 CHILD_MAX 25 (minimum maximum)
```

Another example:

377 Base standard (POSIX.1-1996): `_POSIX_NGROUPS_MAX` 0  
 378 Base standard: `NGROUPS_MAX` minimum maximum `_POSIX_NGROUP_MAX`  
 379 FIPS profile/SUSv2 `NGROUPS_MAX` 8

380 A profile may lower a minimum maximum below the equivalent `_POSIX` value:

381 Base standard: `_POSIX_foo_MAX` Z  
 382 Base standard: `foo_MAX` `_POSIX_foo_MAX`  
 383 profile standard : `foo_MAX` X (X can be less than, equal to,  
 384 or greater than `_POSIX_foo_MAX`)

385 In this case an implementation conforming to the profile may not conform to the base standard,  
 386 but an implementation to the base standard will conform to the profile.

387 IEEE Std 1003.1-2001/Cor 2-2004, item XBD/TC2/D6/2 is applied, clarifying the wording under 2  
 388 the Realtime Option Group when `_POSIX_PRIORITIZED_IO` is supported to take threads into 2  
 389 account. 2

#### 390 A.2.1.6 Options

391 The final subsections within *Implementation Conformance* list the core options within  
 392 IEEE Std 1003.1-2001. This includes both options for the System Interfaces volume of  
 393 IEEE Std 1003.1-2001 and the Shell and Utilities volume of IEEE Std 1003.1-2001.

### 394 A.2.2 Application Conformance

395 These definitions guide users or adaptors of applications in determining on which  
 396 implementations an application will run and how much adaptation would be required to make  
 397 it run on others. These definitions are modeled after related ones in the ISO C standard.

398 POSIX.1 occasionally uses the expressions “portable application” or “conforming application”.  
 399 As they are used, these are synonyms for any of these terms. The differences between the classes  
 400 of application conformance relate to the requirements for other standards, the options supported  
 401 (such as the XSI extension) or, in the case of the Conforming POSIX.1 Application Using  
 402 Extensions, to implementation extensions. When one of the less explicit expressions is used, it  
 403 should be apparent from the context of the discussion which of the more explicit names is  
 404 appropriate

#### 405 A.2.2.1 Strictly Conforming POSIX Application

406 This definition is analogous to that of an ISO C standard “conforming program”.

407 The major difference between a Strictly Conforming POSIX Application and an ISO C standard  
 408 strictly conforming program is that the latter is not allowed to use features of POSIX that are not  
 409 in the ISO C standard.

#### 410 A.2.2.2 Conforming POSIX Application

411 Examples of <National Bodies> include ANSI, BSI, and AFNOR.

#### 412 A.2.2.3 Conforming POSIX Application Using Extensions

413 Due to possible requirements for configuration or implementation characteristics in excess of the  
 414 specifications in <limits.h> or related to the hardware (such as array size or file space), not every  
 415 Conforming POSIX Application Using Extensions will run on every conforming  
 416 implementation.



#### 417 A.2.2.4 *Strictly Conforming XSI Application*

418 This is intended to be upwards-compatible with the definition of a Strictly Conforming POSIX  
419 Application, with the addition of the facilities and functionality included in the XSI extension.

#### 420 A.2.2.5 *Conforming XSI Application Using Extensions*

421 Such applications may use extensions beyond the facilities defined by IEEE Std 1003.1-2001  
422 including the XSI extension, but need to document the additional requirements.

### 423 A.2.3 **Language-Dependent Services for the C Programming Language**

424 POSIX.1 is, for historical reasons, both a specification of an operating system interface, shell and  
425 utilities, and a C binding for that specification. Efforts had been previously undertaken to  
426 generate a language-independent specification; however, that had failed, and the fact that the  
427 ISO C standard is the *de facto* primary language on POSIX and the UNIX system makes this a  
428 necessary and workable situation.

#### 429 A.2.4 **Other Language-Related Specifications**

430 There is no additional rationale provided for this section.

### 431 A.3 **Definitions**

432 The definitions in this section are stated so that they can be used as exact substitutes for the  
433 terms in text. They should not contain requirements or cross-references to sections within  
434 IEEE Std 1003.1-2001; that is accomplished by using an informative note. In addition, the term  
435 should not be included in its own definition. Where requirements or descriptions need to be  
436 addressed but cannot be included in the definitions, due to not meeting the above criteria, these  
437 occur in the General Concepts chapter.

438 In this revision, the definitions have been reworked extensively to meet style requirements and  
439 to include terms from the base documents (see the Scope).

440 Many of these definitions are necessarily circular, and some of the terms (such as “process”) are  
441 variants of basic computing science terms that are inherently hard to define. Where some  
442 definitions are more conceptual and contain requirements, these appear in the General Concepts  
443 chapter. Those listed in this section appear in an alphabetical glossary format of terms.

444 Some definitions must allow extension to cover terms or facilities that are not explicitly  
445 mentioned in IEEE Std 1003.1-2001. For example, the definition of “Extended Security Controls”  
446 permits implementations beyond those defined in IEEE Std 1003.1-2001.

447 Some terms in the following list of notes do not appear in IEEE Std 1003.1-2001; these are  
448 marked suffixed with an asterisk (\*). Many of them have been specifically excluded from 1  
449 IEEE Std 1003.1-2001 because they concern system administration, implementation, or other  
450 issues that are not specific to the programming interface. Those are marked with a reason, such  
451 as “implementation-defined”.

## Appropriate Privileges

One of the fundamental security problems with many historical UNIX systems has been that the privilege mechanism is monolithic—a user has either no privileges or *all* privileges. Thus, a successful “trojan horse” attack on a privileged process defeats all security provisions. Therefore, POSIX.1 allows more granular privilege mechanisms to be defined. For many historical implementations of the UNIX system, the presence of the term “appropriate privileges” in POSIX.1 may be understood as a synonym for “superuser” (UID 0). However, other systems have emerged where this is not the case and each discrete controllable action has *appropriate privileges* associated with it. Because this mechanism is implementation-defined, it must be described in the conformance document. Although that description affects several parts of POSIX.1 where the term “appropriate privilege” is used, because the term “implementation-defined” only appears here, the description of the entire mechanism and its effects on these other sections belongs in this equivalent section of the conformance document. This is especially convenient for implementations with a single mechanism that applies in all areas, since it only needs to be described once.

## Byte

The restriction that a byte is now exactly eight bits was a conscious decision by the standard developers. It came about due to a combination of factors, primarily the use of the type `int8_t` within the networking functions and the alignment with the ISO/IEC 9899:1999 standard, where the `intN_t` types are now defined.

According to the ISO/IEC 9899:1999 standard:

- The `[u]intN_t` types must be two's complement with no padding bits and no illegal values.
- All types (apart from bit fields, which are not relevant here) must occupy an integral number of bytes.
- If a type with width  $W$  occupies  $B$  bytes with  $C$  bits per byte ( $C$  is the value of `{CHAR_BIT}`), then it has  $P$  padding bits where  $P+W=B*C$ .
- Therefore, for `int8_t`  $P=0$ ,  $W=8$ . Since  $B \geq 1$ ,  $C \geq 8$ , the only solution is  $B=1$ ,  $C=8$ .

The standard developers also felt that this was not an undue restriction for the current state-of-the-art for this version of IEEE Std 1003.1, but recognize that if industry trends continue, a wider character type may be required in the future.

## Character

The term “character” is used to mean a sequence of one or more bytes representing a single graphic symbol. The deviation in the exact text of the ISO C standard definition for “byte” meets the intent of the rationale of the ISO C standard also clears up the ambiguity raised by the term “basic execution character set”. The octet-minimum requirement is a reflection of the `{CHAR_BIT}` value.

## Child Process

IEEE Std 1003.1-2001/Cor 2-2004, item XBD/TC2/D6/3 is applied, adding the `vfork()` function to those listed.

2  
2

**Clock Tick**

The ISO C standard defines a similar interval for use by the *clock()* function. There is no requirement that these intervals be the same. In historical implementations these intervals are different.

**Command**

The terms “command” and “utility” are related but have distinct meanings. Command is defined as “a directive to a shell to perform a specific task”. The directive can be in the form of a single utility name (for example, *ls*), or the directive can take the form of a compound command (for example, “*ls | grep name | pr*”). A utility is a program that can be called by name from a shell. Issuing only the name of the utility to a shell is the equivalent of a one-word command. A utility may be invoked as a separate program that executes in a different process than the command language interpreter, or it may be implemented as a part of the command language interpreter. For example, the *echo* command (the directive to perform a specific task) may be implemented such that the *echo* utility (the logic that performs the task of echoing) is in a separate program; therefore, it is executed in a process that is different from the command language interpreter. Conversely, the logic that performs the *echo* utility could be built into the command language interpreter; therefore, it could execute in the same process as the command language interpreter.

The terms “tool” and “application” can be thought of as being synonymous with “utility” from the perspective of the operating system kernel. Tools, applications, and utilities historically have run, typically, in processes above the kernel level. Tools and utilities historically have been a part of the operating system non-kernel code and have performed system-related functions, such as listing directory contents, checking file systems, repairing file systems, or extracting system status information. Applications have not generally been a part of the operating system, and they perform non-system-related functions, such as word processing, architectural design, mechanical design, workstation publishing, or financial analysis. Utilities have most frequently been provided by the operating system distributor, applications by third-party software distributors, or by the users themselves. Nevertheless, IEEE Std 1003.1-2001 does not differentiate between tools, utilities, and applications when it comes to receiving services from the system, a shell, or the standard utilities. (For example, the *xargs* utility invokes another utility; it would be of fairly limited usefulness if the users could not run their own applications in place of the standard utilities.) Utilities are not applications in the sense that they are not themselves subject to the restrictions of IEEE Std 1003.1-2001 or any other standard—there is no requirement for *grep*, *stty*, or any of the utilities defined here to be any of the classes of conforming applications.

**Column Positions**

In most 1-byte character sets, such as ASCII, the concept of column positions is identical to character positions and to bytes. Therefore, it has been historically acceptable for some implementations to describe line folding or tab stops or table column alignment in terms of bytes or character positions. Other character sets pose complications, as they can have internal representations longer than one octet and they can have display characters that have different widths on the terminal screen or printer.

In IEEE Std 1003.1-2001 the term “column positions” has been defined to mean character—not byte—positions in input files (such as “column position 7 of the FORTRAN input”). Output files describe the column position in terms of the display width of the narrowest printable character in the character set, adjusted to fit the characteristics of the output device. It is very possible that *n* column positions will not be able to hold *n* characters in some character sets, unless all of those characters are of the narrowest width. It is assumed that the implementation is aware of the

width of the various characters, deriving this information from the value of *LC\_CTYPE*, and thus can determine how many column positions to allot for each character in those utilities where it is important.

The term “column position” was used instead of the more natural “column” because the latter is frequently used in the different contexts of columns of figures, columns of table values, and so on. Wherever confusion might result, these latter types of columns are referred to as “text columns”.

### Controlling Terminal

The question of which of possibly several special files referring to the terminal is meant is not addressed in POSIX.1. The filename */dev/tty* is a synonym for the controlling terminal associated with a process.

### Device Number\*

The concept is handled in *stat()* as *ID of device*.

### Direct I/O

Historically, direct I/O refers to the system bypassing intermediate buffering, but may be extended to cover implementation-defined optimizations.

### Directory

The format of the directory file is implementation-defined and differs radically between System V and 4.3 BSD. However, routines (derived from 4.3 BSD) for accessing directories and certain constraints on the format of the information returned by those routines are described in the *<dirent.h>* header.

### Directory Entry

Throughout IEEE Std 1003.1-2001, the term “link” is used (about the *link()* function, for example) in describing the objects that point to files from directories.

### Display

The Shell and Utilities volume of IEEE Std 1003.1-2001 assigns precise requirements for the terms “display” and “write”. Some historical systems have chosen to implement certain utilities without using the traditional file descriptor model. For example, the *vi* editor might employ direct screen memory updates on a personal computer, rather than a *write()* system call. An instance of user prompting might appear in a dialog box, rather than with standard error. When the Shell and Utilities volume of IEEE Std 1003.1-2001 uses the term “display”, the method of outputting to the terminal is unspecified; many historical implementations use *termcap* or *terminfo*, but this is not a requirement. The term “write” is used when the Shell and Utilities volume of IEEE Std 1003.1-2001 mandates that a file descriptor be used and that the output can be redirected. However, it is assumed that when the writing is directly to the terminal (it has not been redirected elsewhere), there is no practical way for a user or test suite to determine whether a file descriptor is being used. Therefore, the use of a file descriptor is mandated only for the redirection case and the implementation is free to use any method when the output is not redirected. The verb *write* is used almost exclusively, with the very few exceptions of those utilities where output redirection need not be supported: *tabs*, *talk*, *tput*, and *vi*.

**Dot**

The symbolic name *dot* is carefully used in POSIX.1 to distinguish the working directory filename from a period or a decimal point.

**Dot-Dot**

Historical implementations permit the use of these filenames without their special meanings. Such use precludes any meaningful use of these filenames by a Conforming POSIX.1 Application. Therefore, such use is considered an extension, the use of which makes an implementation non-conforming; see also Section A.4.11 (on page 39).

**Epoch**

Historically, the origin of UNIX system time was referred to as “00:00:00 GMT, January 1, 1970”. Greenwich Mean Time is actually not a term acknowledged by the international standards community; therefore, this term, “Epoch”, is used to abbreviate the reference to the actual standard, Coordinated Universal Time.

**FIFO Special File**

See **Pipe** (on page 25).

**File**

It is permissible for an implementation-defined file type to be non-readable or non-writable.

**File Classes**

These classes correspond to the historical sets of permission bits. The classes are general to allow implementations flexibility in expanding the access mechanism for more stringent security environments. Note that a process is in one and only one class, so there is no ambiguity.

**Filename**

At the present time, the primary responsibility for truncating filenames containing multi-byte characters must reside with the application. Some industry groups involved in internationalization believe that in the future the responsibility must reside with the kernel. For the moment, a clearer understanding of the implications of making the kernel responsible for truncation of multi-byte filenames is needed.

Character-level truncation was not adopted because there is no support in POSIX.1 that advises how the kernel distinguishes between single and multi-byte characters. Until that time, it must be incumbent upon application writers to determine where multi-byte characters must be truncated.

**File System**

Historically, the meaning of this term has been overloaded with two meanings: that of the complete file hierarchy, and that of a mountable subset of that hierarchy; that is, a mounted file system. POSIX.1 uses the term “file system” in the second sense, except that it is limited to the scope of a process (and a process’ root directory). This usage also clarifies the domain in which a file serial number is unique.

**Graphic Character**

This definition is made available for those definitions (in particular, *TZ*) which must exclude control characters.

**Group Database**

IEEE Std 1003.1-2001/Cor 2-2004, item XBD/TC2/D6/4 is applied, removing the words “of implementation-defined format”. See **User Database** (on page 33). 2 2

**Group File\***

Implementation-defined; see **User Database** (on page 33).

**Historical Implementations\***

This refers to previously existing implementations of programming interfaces and operating systems that are related to the interface specified by POSIX.1.

**Hosted Implementation\***

This refers to a POSIX.1 implementation that is accomplished through interfaces from the POSIX.1 services to some alternate form of operating system kernel services. Note that the line between a hosted implementation and a native implementation is blurred, since most implementations will provide some services directly from the kernel and others through some indirect path. (For example, *fopen()* might use *open()*; or *mkfifo()* might use *mknod()*.) There is no necessary relationship between the type of implementation and its correctness, performance, and/or reliability.

**Implementation\***

This term is generally used instead of its synonym, “system”, to emphasize the consequences of decisions to be made by system implementors. Perhaps if no options or extensions to POSIX.1 were allowed, this usage would not have occurred.

The term “specific implementation” is sometimes used as a synonym for “implementation”. This should not be interpreted too narrowly; both terms can represent a relatively broad group of systems. For example, a hardware vendor could market a very wide selection of systems that all used the same instruction set, with some systems desktop models and others large multi-user minicomputers. This wide range would probably share a common POSIX.1 operating system, allowing an application compiled for one to be used on any of the others; this is a [*specific*] *implementation*. However, such a wide range of machines probably has some differences between the models. Some may have different clock rates, different file systems, different resource limits, different network connections, and so on, depending on their sizes or intended usages. Even on two identical machines, the system administrators may configure them differently. Each of these different systems is known by the term “a specific instance of a specific implementation”. This term is only used in the portions of POSIX.1 dealing with runtime queries: *sysconf()* and *pathconf()*.

**Incomplete Pathname\***

Absolute pathname has been adequately defined.

**Job Control**

In order to understand the job control facilities in POSIX.1 it is useful to understand how they are used by a job control-cognizant shell to create the user interface effect of job control.

While the job control facilities supplied by POSIX.1 can, in theory, support different types of interactive job control interfaces supplied by different types of shells, there was historically one particular interface that was most common when the standard was originally developed (provided by BSD C Shell).

This discussion describes that interface as a means of illustrating how the POSIX.1 job control facilities can be used. 1

Job control allows users to selectively stop (suspend) the execution of processes and continue (resume) their execution at a later point. The user typically employs this facility via the interactive interface jointly supplied by the terminal I/O driver and a command interpreter (shell).

The user can launch jobs (command pipelines) in either the foreground or background. When launched in the foreground, the shell waits for the job to complete before prompting for additional commands. When launched in the background, the shell does not wait, but immediately prompts for new commands.

If the user launches a job in the foreground and subsequently regrets this, the user can type the suspend character (typically set to <control>-Z), which causes the foreground job to stop and the shell to begin prompting for new commands. The stopped job can be continued by the user (via special shell commands) either as a foreground job or as a background job. Background jobs can also be moved into the foreground via shell commands.

If a background job attempts to access the login terminal (controlling terminal), it is stopped by the terminal driver and the shell is notified, which, in turn, notifies the user. (Terminal access includes *read()* and certain terminal control functions, and conditionally includes *write()*.) The user can continue the stopped job in the foreground, thus allowing the terminal access to succeed in an orderly fashion. After the terminal access succeeds, the user can optionally move the job into the background via the suspend character and shell commands.

**Implementing Job Control Shells**

The interactive interface described previously can be accomplished using the POSIX.1 job control facilities in the following way.

The key feature necessary to provide job control is a way to group processes into jobs. This grouping is necessary in order to direct signals to a single job and also to identify which job is in the foreground. (There is at most one job that is in the foreground on any controlling terminal at a time.)

The concept of process groups is used to provide this grouping. The shell places each job in a separate process group via the *setpgid()* function. To do this, the *setpgid()* function is invoked by the shell for each process in the job. It is actually useful to invoke *setpgid()* twice for each process: once in the child process, after calling *fork()* to create the process, but before calling one of the *exec* family of functions to begin execution of the program, and once in the parent shell process, after calling *fork()* to create the child. The redundant invocation avoids a race condition by ensuring that the child process is placed into the new process group before either the parent or the child relies on this being the case. The process group ID for the job is selected by the shell

to be equal to the process ID of one of the processes in the job. Some shells choose to make one process in the job be the parent of the other processes in the job (if any). Other shells (for example, the C Shell) choose to make themselves the parent of all processes in the pipeline (job). In order to support this latter case, the *setpgid()* function accepts a process group ID parameter since the correct process group ID cannot be inherited from the shell. The shell itself is considered to be a job and is the sole process in its own process group.

The shell also controls which job is currently in the foreground. A foreground and background job differ in two ways: the shell waits for a foreground command to complete (or stop) before continuing to read new commands, and the terminal I/O driver inhibits terminal access by background jobs (causing the processes to stop). Thus, the shell must work cooperatively with the terminal I/O driver and have a common understanding of which job is currently in the foreground. It is the user who decides which command should be currently in the foreground, and the user informs the shell via shell commands. The shell, in turn, informs the terminal I/O driver via the *tcsetpgrp()* function. This indicates to the terminal I/O driver the process group ID of the foreground process group (job). When the current foreground job either stops or terminates, the shell places itself in the foreground via *tcsetpgrp()* before prompting for additional commands. Note that when a job is created the new process group begins as a background process group. It requires an explicit act of the shell via *tcsetpgrp()* to move a process group (job) into the foreground.

When a process in a job stops or terminates, its parent (for example, the shell) receives synchronous notification by calling the *waitpid()* function with the WUNTRACED flag set. Asynchronous notification is also provided when the parent establishes a signal handler for SIGCHLD and does not specify the SA\_NOCLDSTOP flag. Usually all processes in a job stop as a unit since the terminal I/O driver always sends job control stop signals to all processes in the process group.

To continue a stopped job, the shell sends the SIGCONT signal to the process group of the job. In addition, if the job is being continued in the foreground, the shell invokes *tcsetpgrp()* to place the job in the foreground before sending SIGCONT. Otherwise, the shell leaves itself in the foreground and reads additional commands.

There is additional flexibility in the POSIX.1 job control facilities that allows deviations from the typical interface. Clearing the TOSTOP terminal flag allows background jobs to perform *write()* functions without stopping. The same effect can be achieved on a per-process basis by having a process set the signal action for SIGTTOU to SIG\_IGN.

Note that the terms “job” and “process group” can be used interchangeably. A login session that is not using the job control facilities can be thought of as a large collection of processes that are all in the same job (process group). Such a login session may have a partial distinction between foreground and background processes; that is, the shell may choose to wait for some processes before continuing to read new commands and may not wait for other processes. However, the terminal I/O driver will consider all these processes to be in the foreground since they are all members of the same process group.

In addition to the basic job control operations already mentioned, a job control-cognizant shell needs to perform the following actions.

When a foreground (not background) job stops, the shell must sample and remember the current terminal settings so that it can restore them later when it continues the stopped job in the foreground (via the *tcgetattr()* and *tcsetattr()* functions).

Because a shell itself can be spawned from a shell, it must take special action to ensure that subshells interact well with their parent shells.



A subshell can be spawned to perform an interactive function (prompting the terminal for commands) or a non-interactive function (reading commands from a file). When operating non-interactively, the job control shell will refrain from performing the job control-specific actions described above. It will behave as a shell that does not support job control. For example, all jobs will be left in the same process group as the shell, which itself remains in the process group established for it by its parent. This allows the shell and its children to be treated as a single job by a parent shell, and they can be affected as a unit by terminal keyboard signals.

An interactive subshell can be spawned from another job control-cognizant shell in either the foreground or background. (For example, from the C Shell, the user can execute the command, `csch &`.) Before the subshell activates job control by calling `setpgid()` to place itself in its own process group and `tcsetpgrp()` to place its new process group in the foreground, it needs to ensure that it has already been placed in the foreground by its parent. (Otherwise, there could be multiple job control shells that simultaneously attempt to control mediation of the terminal.) To determine this, the shell retrieves its own process group via `getpgrp()` and the process group of the current foreground job via `tcgetpgrp()`. If these are not equal, the shell sends SIGTTIN to its own process group, causing itself to stop. When continued later by its parent, the shell repeats the process group check. When the process groups finally match, the shell is in the foreground and it can proceed to take control. After this point, the shell ignores all the job control stop signals so that it does not inadvertently stop itself.

### *Implementing Job Control Applications*

Most applications do not need to be aware of job control signals and operations; the intuitively correct behavior happens by default. However, sometimes an application can inadvertently interfere with normal job control processing, or an application may choose to overtly effect job control in cooperation with normal shell procedures.

An application can inadvertently subvert job control processing by “blindly” altering the handling of signals. A common application error is to learn how many signals the system supports and to ignore or catch them all. Such an application makes the assumption that it does not know what this signal is, but knows the right handling action for it. The system may initialize the handling of job control stop signals so that they are being ignored. This allows shells that do not support job control to inherit and propagate these settings and hence to be immune to stop signals. A job control shell will set the handling to the default action and propagate this, allowing processes to stop. In doing so, the job control shell is taking responsibility for restarting the stopped applications. If an application wishes to catch the stop signals itself, it should first determine their inherited handling states. If a stop signal is being ignored, the application should continue to ignore it. This is directly analogous to the recommended handling of SIGINT described in the referenced UNIX Programmer's Manual.

If an application is reading the terminal and has disabled the interpretation of special characters (by clearing the ISIG flag), the terminal I/O driver will not send SIGTSTP when the suspend character is typed. Such an application can simulate the effect of the suspend character by recognizing it and sending SIGTSTP to its process group as the terminal driver would have done. Note that the signal is sent to the process group, not just to the application itself; this ensures that other processes in the job also stop. (Note also that other processes in the job could be children, siblings, or even ancestors.) Applications should not assume that the suspend character is `<control>-Z` (or any particular value); they should retrieve the current setting at startup.

### Implementing Job Control Systems

The intent in adding 4.2 BSD-style job control functionality was to adopt the necessary 4.2 BSD programmatic interface with only minimal changes to resolve syntactic or semantic conflicts with System V or to close recognized security holes. The goal was to maximize the ease of providing both conforming implementations and Conforming POSIX.1 Applications.

It is only useful for a process to be affected by job control signals if it is the descendant of a job control shell. Otherwise, there will be nothing that continues the stopped process.

POSIX.1 does not specify how controlling terminal access is affected by a user logging out (that is, by a controlling process terminating). 4.2 BSD uses the *vhangup()* function to prevent any access to the controlling terminal through file descriptors opened prior to logout. System V does not prevent controlling terminal access through file descriptors opened prior to logout (except for the case of the special file, */dev/tty*). Some implementations choose to make processes immune from job control after logout (that is, such processes are always treated as if in the foreground); other implementations continue to enforce foreground/background checks after logout. Therefore, a Conforming POSIX.1 Application should not attempt to access the controlling terminal after logout since such access is unreliable. If an implementation chooses to deny access to a controlling terminal after its controlling process exits, POSIX.1 requires a certain type of behavior (see **Controlling Terminal** (on page 16)).

### Kernel\*

See **System Call\*** (on page 31).

### Library Routine\*

See **System Call\*** (on page 31).

### Logical Device\*

Implementation-defined.

### Map

The definition of map is included to clarify the usage of mapped pages in the description of the behavior of process memory locking.

### Memory-Resident

The term “memory-resident” is historically understood to mean that the so-called resident pages are actually present in the physical memory of the computer system and are immune from swapping, paging, copy-on-write faults, and so on. This is the actual intent of IEEE Std 1003.1-2001 in the process memory locking section for implementations where this is logical. But for some implementations—primarily mainframes—actually locking pages into primary storage is not advantageous to other system objectives, such as maximizing throughput. For such implementations, memory locking is a “hint” to the implementation that the application wishes to avoid situations that would cause long latencies in accessing memory. Furthermore, there are other implementation-defined issues with minimizing memory access latencies that “memory residency” does not address—such as MMU reload faults. The definition attempts to accommodate various implementations while allowing conforming applications to specify to the implementation that they want or need the best memory access times that the implementation can provide.

**Memory Object\***

The term “memory object” usually implies shared memory. If the object is the same as a filename in the file system name space of the implementation, it is expected that the data written into the memory object be preserved on disk. A memory object may also apply to a physical device on an implementation. In this case, writes to the memory object are sent to the controller for the device and reads result in control registers being returned.

**Mount Point\***

The directory on which a “mounted file system” is mounted. This term, like *mount()* and *umount()*, was not included because it was implementation-defined.

See **File System** (on page 17).

**Name**

There are no explicit limits in IEEE Std 1003.1-2001 on the sizes of names, words (see the definition of word in the Base Definitions volume of IEEE Std 1003.1-2001), lines, or other objects. However, other implicit limits do apply: shell script lines produced by many of the standard utilities cannot exceed {LINE\_MAX} and the sum of exported variables comes under the {ARG\_MAX} limit. Historical shells dynamically allocate memory for names and words and parse incoming lines a character at a time. Lines cannot have an arbitrary {LINE\_MAX} limit because of historical practice, such as makefiles, where *make* removes the <newline>s associated with the commands for a target and presents the shell with one very long line. The text on INPUT FILES in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section 1.11, Utility Description Defaults does allow a shell to run out of memory, but it cannot have arbitrary programming limits.

**Native Implementation\***

This refers to an implementation of POSIX.1 that interfaces directly to an operating system kernel; see also *hosted implementation*. A similar concept is a native UNIX system, which would be a kernel derived from one of the original UNIX system products. 1

**Nice Value**

This definition is not intended to suggest that all processes in a system have priorities that are comparable. Scheduling policy extensions, such as adding realtime priorities, make the notion of a single underlying priority for all scheduling policies problematic. Some implementations may implement the features related to *nice* to affect all processes on the system, others to affect just the general time-sharing activities implied by IEEE Std 1003.1-2001, and others may have no effect at all. Because of the use of “implementation-defined” in *nice* and *renice*, a wide range of implementation strategies is possible.

**Open File Description**

An “open file description”, as it is currently named, describes how a file is being accessed. What is currently called a “file descriptor” is actually just an identifier or “handle”; it does not actually describe anything.

The following alternate names were discussed:

- For “open file description”:  
“open instance”, “file access description”, “open file information”, and “file access information”.
- For “file descriptor”:  
“file handle”, “file number” (cf., *fileno()*). Some historical implementations use the term “file table entry”.

**Orphaned Process Group**

Historical implementations have a concept of an orphaned process, which is a process whose parent process has exited. When job control is in use, it is necessary to prevent processes from being stopped in response to interactions with the terminal after they no longer are controlled by a job control-cognizant program. Because signals generated by the terminal are sent to a process group and not to individual processes, and because a signal may be provoked by a process that is not orphaned, but sent to another process that is orphaned, it is necessary to define an orphaned process group. The definition assumes that a process group will be manipulated as a group and that the job control-cognizant process controlling the group is outside of the group and is the parent of at least one process in the group (so that state changes may be reported via *waitpid()*). Therefore, a group is considered to be controlled as long as at least one process in the group has a parent that is outside of the process group, but within the session.

This definition of orphaned process groups ensures that a session leader’s process group is always considered to be orphaned, and thus it is prevented from stopping in response to terminal signals.

**Page**

The term “page” is defined to support the description of the behavior of memory mapping for shared memory and memory mapped files, and the description of the behavior of process memory locking. It is not intended to imply that shared memory/file mapping and memory locking are applicable only to “paged” architectures. For the purposes of IEEE Std 1003.1-2001, whatever the granularity on which an architecture supports mapping or locking, this is considered to be a “page”. If an architecture cannot support the memory mapping or locking functions specified by IEEE Std 1003.1-2001 on any granularity, then these options will not be implemented on the architecture.

**Passwd File\***

Implementation-defined; see **User Database** (on page 33).

902	<b>Parent Directory</b>	
903	There may be more than one directory entry pointing to a given directory in some	
904	implementations. The wording here identifies that exactly one of those is the parent directory. In	
905	pathname resolution, dot-dot is identified as the way that the unique directory is identified.	
906	(That is, the parent directory is the one to which dot-dot points.) In the case of a remote file	
907	system, if the same file system is mounted several times, it would appear as if they were distinct	
908	file systems (with interesting synchronization properties).	
909	<b>Pipe</b>	
910	It proved convenient to define a pipe as a special case of a FIFO, even though historically the	
911	latter was not introduced until System III and does not exist at all in 4.3 BSD.	
912	<b>Portable Filename Character Set</b>	
913	The encoding of this character set is not specified—specifically, ASCII is not required. But the	
914	implementation must provide a unique character code for each of the printable graphics	
915	specified by POSIX.1; see also Section A.4.6 (on page 35).	
916	Situations where characters beyond the portable filename character set (or historically ASCII or	
917	the ISO/IEC 646:1991 standard) would be used (in a context where the portable filename	
918	character set or the ISO/IEC 646:1991 standard is required by POSIX.1) are expected to be	
919	common. Although such a situation renders the use technically non-compliant, mutual	
920	agreement among the users of an extended character set will make such use portable between	
921	those users. Such a mutual agreement could be formalized as an optional extension to POSIX.1.	
922	(Making it required would eliminate too many possible systems, as even those systems using the	
923	ISO/IEC 646:1991 standard as a base character set extend their character sets for Western	
924	Europe and the rest of the world in different ways.)	
925	Nothing in POSIX.1 is intended to preclude the use of extended characters where interchange is	
926	not required or where mutual agreement is obtained. It has been suggested that in several places	
927	“should” be used instead of “shall”. Because (in the worst case) use of any character beyond the	
928	portable filename character set would render the program or data not portable to all possible	
929	systems, no extensions are permitted in this context.	
930	<b>Process Lifetime</b>	2
931	IEEE Std 1003.1-2001/Cor 2-2004, item XBD/TC2/D6/5 is applied, adding <i>fork()</i> , <i>posix_spawn()</i> ,	2
932	<i>posix_spawnnp()</i> , and <i>vfork()</i> to the list of functions.	2
933	<b>Process Termination</b>	2
934	IEEE Std 1003.1-2001/Cor 2-2004, item XBD/TC2/D6/6 is applied, rewording the definition to	2
935	address the “passive exit” on termination of the last thread or the <i>_Exit()</i> function.	2
936	<b>Regular File</b>	
937	POSIX.1 does not intend to preclude the addition of structuring data (for example, record	
938	lengths) in the file, as long as such data is not visible to an application that uses the features	
939	described in POSIX.1.	

**Root Directory**

This definition permits the operation of *chroot()*, even though that function is not in POSIX.1; see also Section A.4.5 (on page 35).

**Root File System\***

Implementation-defined.

**Root of a File System\***

Implementation-defined; see **Mount Point\*** (on page 23).

**Signal**

The definition implies a double meaning for the term. Although a signal is an event, common usage implies that a signal is an identifier of the class of event.

**Superuser\***

This concept, with great historical significance to UNIX system users, has been replaced with the notion of appropriate privileges.

**Supplementary Group ID**

The POSIX.1-1990 standard is inconsistent in its treatment of supplementary groups. The definition of supplementary group ID explicitly permits the effective group ID to be included in the set, but wording in the description of the *setuid()* and *setgid()* functions states: “Any supplementary group IDs of the calling process remain unchanged by these function calls”. In the case of *setgid()* this contradicts that definition. In addition, some felt that the unspecified behavior in the definition of supplementary group IDs adds unnecessary portability problems. The standard developers considered several solutions to this problem:

1. Reword the description of *setgid()* to permit it to change the supplementary group IDs to reflect the new effective group ID. A problem with this is that it adds more “may”s to the wording and does not address the portability problems of this optional behavior.
2. Mandate the inclusion of the effective group ID in the supplementary set (giving {NGROUPS\_MAX} a minimum value of 1). This is the behavior of 4.4 BSD. In that system, the effective group ID is the first element of the array of supplementary group IDs (there is no separate copy stored, and changes to the effective group ID are made only in the supplementary group set). By convention, the initial value of the effective group ID is duplicated elsewhere in the array so that the initial value is not lost when executing a set-group-ID program.
3. Change the definition of supplementary group ID to exclude the effective group ID and specify that the effective group ID does not change the set of supplementary group IDs. This is the behavior of 4.2 BSD, 4.3 BSD, and System V Release 4.
4. Change the definition of supplementary group ID to exclude the effective group ID, and require that *getgroups()* return the union of the effective group ID and the supplementary group IDs.
5. Change the definition of {NGROUPS\_MAX} to be one more than the number of supplementary group IDs, so it continues to be the number of values returned by *getgroups()* and existing applications continue to work. This alternative is effectively the same as the second (and might actually have the same implementation).

The standard developers decided to permit either 2 or 3. The effective group ID is orthogonal to the set of supplementary group IDs, and it is implementation-defined whether *getgroups()* returns this. If the effective group ID is returned with the set of supplementary group IDs, then all changes to the effective group ID affect the supplementary group set returned by *getgroups()*. It is permissible to eliminate duplicates from the list returned by *getgroups()*. However, if a group ID is contained in the set of supplementary group IDs, setting the group ID to that value and then to a different value should not remove that value from the supplementary group IDs.

The definition of supplementary group IDs has been changed to not include the effective group ID. This simplifies permanent rationale and makes the relevant functions easier to understand. The *getgroups()* function has been modified so that it can, on an implementation-defined basis, return the effective group ID. By making this change, functions that modify the effective group ID do not need to discuss adding to the supplementary group list; the only view into the supplementary group list that the application writer has is through the *getgroups()* function.

## Symbolic Link

Many implementations associate no attributes, including ownership with symbolic links. Security experts encouraged consideration for defining these attributes as optional. Consideration was given to changing *utime()* to allow modification of the times for a symbolic link, or as an alternative adding an *lutime()* interface. Modifications to *chown()* were also considered: allow changing symbolic link ownership or alternatively adding *lchown()*. As a result of alignment with the Single UNIX Specification, the *lchown()* function is included as part of the XSI extension and XSI-conformant systems may support an owner and a group associated with a symbolic link. There was no consensus to define further attributes for symbolic links, and for systems not supporting the XSI extension only the file type bits in the *st\_mode* member and the *st\_size* member of the *stat* structure are required to be applicable to symbolic links.

Historical implementations were followed when determining which interfaces should apply to symbolic links. Interfaces that historically followed symbolic links include *chmod()*, *link()*, and *utime()*. Interfaces that historically do not follow symbolic links include *chown()*, *lstat()*, *readlink()*, *rename()*, *remove()*, *rmdir()*, and *unlink()*. IEEE Std 1003.1-2001 deviates from historical practice only in the case of *chown()*. Because there is no requirement for systems not supporting the XSI extension that there is an association of ownership with symbolic links, there was no interface in the base standard to change ownership. In addition, other implementations of symbolic links have modified *chown()* to follow symbolic links.

In the case of symbolic links, IEEE Std 1003.1-2001 states that a trailing slash is considered to be the final component of a pathname rather than the pathname component that preceded it. This is the behavior of historical implementations. For example, for */a/b* and */a/b/*, if */a/b* is a symbolic link to a directory, then */a/b* refers to the symbolic link, and */a/b/* is the same as */a/b/.*, which is the directory to which the symbolic link points.

For multi-level security purposes, it is possible to have the link read mode govern permission for the *readlink()* function. It is also possible that the read permissions of the directory containing the link be used for this purpose. Implementations may choose to use either of these methods; however, this is not current practice and neither method is specified.

Several reasons were advanced for requiring that when a symbolic link is used as the source argument to the *link()* function, the resulting link will apply to the file named by the contents of the symbolic link rather than to the symbolic link itself. This is the case in historical implementations. This action was preferred, as it supported the traditional idea of persistence with respect to the target of a hard link. This decision is appropriate in light of a previous decision not to require association of attributes with symbolic links, thereby allowing implementations which do not use inodes. Opposition centered on the lack of symmetry on the

part of the *link()* and *unlink()* function pair with respect to symbolic links.

Because a symbolic link and its referenced object coexist in the file system name space, confusion can arise in distinguishing between the link itself and the referenced object. Historically, utilities and system calls have adopted their own link following conventions in a somewhat *ad hoc* fashion. Rules for a uniform approach are outlined here, although historical practice has been adhered to as much as was possible. To promote consistent system use, user-written utilities are encouraged to follow these same rules.

Symbolic links are handled either by operating on the link itself, or by operating on the object referenced by the link. In the latter case, an application or system call is said to “follow” the link. Symbolic links may reference other symbolic links, in which case links are dereferenced until an object that is not a symbolic link is found, a symbolic link that references a file that does not exist is found, or a loop is detected. (Current implementations do not detect loops, but have a limit on the number of symbolic links that they will dereference before declaring it an error.)

There are four domains for which default symbolic link policy is established in a system. In almost all cases, there are utility options that override this default behavior. The four domains are as follows:

1. Symbolic links specified to system calls that take filename arguments
2. Symbolic links specified as command line filename arguments to utilities that are not performing a traversal of a file hierarchy
3. Symbolic links referencing files not of type directory, specified to utilities that are performing a traversal of a file hierarchy
4. Symbolic links referencing files of type directory, specified to utilities that are performing a traversal of a file hierarchy

#### *First Domain*

The first domain is considered in earlier rationale.

#### *Second Domain*

The reason this category is restricted to utilities that are not traversing the file hierarchy is that some standard utilities take an option that specifies a hierarchical traversal, but by default operate on the arguments themselves. Generally, users specifying the option for a file hierarchy traversal wish to operate on a single, physical hierarchy, and therefore symbolic links, which may reference files outside of the hierarchy, are ignored. For example, *chown owner file* is a different operation from the same command with the *-R* option specified. In this example, the behavior of the command *chown owner file* is described here, while the behavior of the command *chown -R owner file* is described in the third and fourth domains.

The general rule is that the utilities in this category follow symbolic links named as arguments.

Exceptions in the second domain are:

- The *mv* and *rm* utilities do not follow symbolic links named as arguments, but respectively attempt to rename or delete them.
- The *ls* utility is also an exception to this rule. For compatibility with historical systems, when the *-R* option is not specified, the *ls* utility follows symbolic links named as arguments if the *-L* option is specified or if the *-F*, *-d*, or *-l* options are not specified. (If the *-L* option is specified, *ls* always follows symbolic links; it is the only utility where the *-L* option affects its behavior even though a tree walk is not being performed.)



All other standard utilities, when not traversing a file hierarchy, always follow symbolic links named as arguments.

Historical practice is that the **-h** option is specified if standard utilities are to act upon symbolic links instead of upon their targets. Examples of commands that have historically had a **-h** option for this purpose are the *chgrp*, *chown*, *file*, and *test* utilities.

### Third Domain

The third domain is symbolic links, referencing files not of type directory, specified to utilities that are performing a traversal of a file hierarchy. (This includes symbolic links specified as command line filename arguments or encountered during the traversal.)

The intention of the Shell and Utilities volume of IEEE Std 1003.1-2001 is that the operation that the utility is performing is applied to the symbolic link itself, if that operation is applicable to symbolic links. The reason that the operation is not required is that symbolic links in some implementations do not have such attributes as a file owner, and therefore the *chown* operation would be meaningless. If symbolic links on the system have an owner, it is the intention that the utility *chown* cause the owner of the symbolic link to change. If symbolic links do not have an owner, the symbolic link should be ignored. Specifically, by default, no change should be made to the file referenced by the symbolic link.

### Fourth Domain

The fourth domain is symbolic links referencing files of type directory, specified to utilities that are performing a traversal of a file hierarchy. (This includes symbolic links specified as command line filename arguments or encountered during the traversal.)

Most standard utilities do not, by default, indirect into the file hierarchy referenced by the symbolic link. (The Shell and Utilities volume of IEEE Std 1003.1-2001 uses the informal term “physical walk” to describe this case. The case where the utility does indirect through the symbolic link is termed a “logical walk”.)

There are three reasons for the default to be a physical walk:

1. With very few exceptions, a physical walk has been the historical default on UNIX systems supporting symbolic links. Because some utilities (that is, *rm*) must default to a physical walk, regardless, changing historical practice in this regard would be confusing to users and needlessly incompatible.
2. For systems where symbolic links have the historical file attributes (that is, *owner*, *group*, *mode*), defaulting to a logical traversal would require the addition of a new option to the commands to modify the attributes of the link itself. This is painful and more complex than the alternatives.
3. There is a security issue with defaulting to a logical walk. Historically, the command *chown -R user file* has been safe for the superuser because *setuid* and *setgid* bits were lost when the ownership of the file was changed. If the walk were logical, changing ownership would no longer be safe because a user might have inserted a symbolic link pointing to any file in the tree. Again, this would necessitate the addition of an option to the commands doing hierarchy traversal to not indirect through the symbolic links, and historical scripts doing recursive walks would instantly become security problems. While this is mostly an issue for system administrators, it is preferable to not have different defaults for different classes of users.

However, the standard developers agreed to leave it unspecified to achieve consensus.

As consistently as possible, users may cause standard utilities performing a file hierarchy traversal to follow any symbolic links named on the command line, regardless of the type of file

they reference, by specifying the **-H** (for half logical) option. This option is intended to make the command line name space look like the logical name space.

As consistently as possible, users may cause standard utilities performing a file hierarchy traversal to follow any symbolic links named on the command line as well as any symbolic links encountered during the traversal, regardless of the type of file they reference, by specifying the **-L** (for logical) option. This option is intended to make the entire name space look like the logical name space.

For consistency, implementors are encouraged to use the **-P** (for “physical”) flag to specify the physical walk in utilities that do logical walks by default for whatever reason. The only standard utilities that require the **-P** option are *cd* and *pwd*; see the note below.

When one or more of the **-H**, **-L**, and **-P** flags can be specified, the last one specified determines the behavior of the utility. This permits users to alias commands so that the default behavior is a logical walk and then override that behavior on the command line.

#### *Exceptions in the Third and Fourth Domains*

The *ls* and *rm* utilities are exceptions to these rules. The *rm* utility never follows symbolic links and does not support the **-H**, **-L**, or **-P** options. Some historical versions of *ls* always followed symbolic links given on the command line whether the **-L** option was specified or not. Historical versions of *ls* did not support the **-H** option. In IEEE Std 1003.1-2001, unless one of the **-H** or **-L** options is specified, the *ls* utility only follows symbolic links to directories that are given as operands. The *ls* utility does not support the **-P** option.

The Shell and Utilities volume of IEEE Std 1003.1-2001 requires that the standard utilities *ls*, *find*, and *pax* detect infinite loops when doing logical walks; that is, a directory, or more commonly a symbolic link, that refers to an ancestor in the current file hierarchy. If the file system itself is corrupted, causing the infinite loop, it may be impossible to recover. Because *find* and *ls* are often used in system administration and security applications, they should attempt to recover and continue as best as they can. The *pax* utility should terminate because the archive it was creating is by definition corrupted. Other, less vital, utilities should probably simply terminate as well. Implementations are strongly encouraged to detect infinite loops in all utilities.

Historical practice is shown in Table A-1 (on page 31). The heading **SVID3** stands for the Third Edition of the System V Interface Definition.

Historically, several shells have had built-in versions of the *pwd* utility. In some of these shells, *pwd* reported the physical path, and in others, the logical path. Implementations of the shell corresponding to IEEE Std 1003.1-2001 must report the logical path by default. Earlier versions of IEEE Std 1003.1-2001 did not require the *pwd* utility to be a built-in utility. Now that *pwd* is required to set an environment variable in the current shell execution environment, it must be a built-in utility.

The *cd* command is required, by default, to treat the filename dot-dot logically. Implementors are required to support the **-P** flag in *cd* so that users can have their current environment handled physically. In 4.3 BSD, *chgrp* during tree traversal changed the group of the symbolic link, not the target. Symbolic links in 4.4 BSD do not have *owner*, *group*, *mode*, or other standard UNIX system file attributes.

1159

Table A-1 Historical Practice for Symbolic Links

Utility	SVID3	4.3 BSD	4.4 BSD	POSIX	Comments
1160 <i>cd</i>				-L	Treat " . . " logically.
1161 <i>cd</i>				-P	Treat " . . " physically.
1162 <i>chgrp</i>			-H	-H	Follow command line symlinks.
1163 <i>chgrp</i>			-h	-L	Follow symlinks.
1164 <i>chgrp</i>	-h			-h	Affect the symlink.
1165 <i>chmod</i>					Affect the symlink.
1166 <i>chmod</i>			-H		Follow command line symlinks.
1167 <i>chmod</i>			-h		Follow symlinks.
1168 <i>chown</i>			-H	-H	Follow command line symlinks.
1169 <i>chown</i>			-h	-L	Follow symlinks.
1170 <i>chown</i>	-h			-h	Affect the symlink.
1171 <i>cp</i>			-H	-H	Follow command line symlinks.
1172 <i>cp</i>			-h	-L	Follow symlinks.
1173 <i>cpio</i>	-L		-L		Follow symlinks.
1174 <i>du</i>			-H	-H	Follow command line symlinks.
1175 <i>du</i>			-h	-L	Follow symlinks.
1176 <i>file</i>	-h			-h	Affect the symlink.
1177 <i>find</i>			-H	-H	Follow command line symlinks.
1178 <i>find</i>			-h	-L	Follow symlinks.
1179 <i>find</i>	-follow		-follow		Follow symlinks.
1180 <i>ln</i>	-s	-s	-s	-s	Create a symbolic link.
1181 <i>ls</i>	-L	-L	-L	-L	Follow symlinks.
1182 <i>ls</i>				-H	Follow command line symlinks.
1183 <i>mv</i>					Operates on the symlink.
1184 <i>pax</i>			-H	-H	Follow command line symlinks.
1185 <i>pax</i>			-h	-L	Follow symlinks.
1186 <i>pwd</i>				-L	Printed path may contain symlinks.
1187 <i>pwd</i>				-P	Printed path will not contain symlinks.
1188 <i>rm</i>					Operates on the symlink.
1189 <i>tar</i>			-H		Follow command line symlinks.
1190 <i>tar</i>		-h	-h		Follow symlinks.
1191 <i>test</i>	-h		-h	-h	Affect the symlink.

1193 **Synchronously-Generated Signal**

1194 Those signals that may be generated synchronously include SIGABRT, SIGBUS, SIGILL, SIGFPE,  
 1195 SIGPIPE, and SIGSEGV.

1196 Any signal sent via the *raise()* function or a *kill()* function targeting the current process is also  
 1197 considered synchronous.

1198 **System Call\***

1199 The distinction between a “system call” and a “library routine” is an implementation detail that  
 1200 may differ between implementations and has thus been excluded from POSIX.1.

1201 See “Interface, Not Implementation” in **Introduction** (on page xiv).

1202	<b>System Console</b>	2
1203	IEEE Std 1003.1-2001/Cor 2-2004, item XBD/TC2/D6/7 is applied, changing from “An	2
1204	implementation-defined device” to “A device”.	2
1205	<b>System Databases</b>	2
1206	IEEE Std 1003.1-2001/Cor 2-2004, item XBD/TC2/D6/9 is applied, rewording the definition to	2
1207	reference the existing definitions for “group database” and “user database”.	2
1208	<b>System Process</b>	2
1209	IEEE Std 1003.1-2001/Cor 2-2004, item XBD/TC2/D6/8 is applied, rewording the definition to	2
1210	remove the requirement for an implementation to define the object.	2
1211	<b>System Reboot</b>	
1212	A “system reboot” is an event initiated by an unspecified circumstance that causes all processes	
1213	(other than special system processes) to be terminated in an implementation-defined manner,	
1214	after which any changes to the state and contents of files created or written to by a Conforming	
1215	POSIX.1 Application prior to the event are implementation-defined.	
1216	IEEE Std 1003.1-2001/Cor 2-2004, item XBD/TC2/D6/10 is applied, changing “An	2
1217	implementation-defined sequence of events” to “An unspecified sequence of events”.	2
1218	<b>Synchronized I/O Data (and File) Integrity Completion</b>	
1219	These terms specify that for synchronized read operations, pending writes must be successfully	
1220	completed before the read operation can complete. This is motivated by two circumstances.	
1221	Firstly, when synchronizing processes can access the same file, but not share common buffers	
1222	(such as for a remote file system), this requirement permits the reading process to guarantee that	
1223	it can read data written remotely. Secondly, having data written synchronously is insufficient to	
1224	guarantee the order with respect to a subsequent write by a reading process, and thus this extra	
1225	read semantic is necessary.	
1226	<b>Text File</b>	
1227	The term “text file” does not prevent the inclusion of control or other non-printable characters	
1228	(other than NUL). Therefore, standard utilities that list text files as inputs or outputs are either	
1229	able to process the special characters or they explicitly describe their limitations within their	
1230	individual descriptions. The definition of “text file” has caused controversy. The only difference	
1231	between text and binary files is that text files have lines of less than {LINE_MAX} bytes, with no	
1232	NUL characters, each terminated by a <newline>. The definition allows a file with a single	
1233	<newline>, but not a totally empty file, to be called a text file. If a file ends with an incomplete	
1234	line it is not strictly a text file by this definition. The <newline> referred to in	
1235	IEEE Std 1003.1-2001 is not some generic line separator, but a single character; files created on	
1236	systems where they use multiple characters for ends of lines are not portable to all conforming	
1237	systems without some translation process unspecified by IEEE Std 1003.1-2001.	

**Thread**

IEEE Std 1003.1-2001 defines a thread to be a flow of control within a process. Each thread has a minimal amount of private state; most of the state associated with a process is shared among all of the threads in the process. While most multi-thread extensions to POSIX have taken this approach, others have made different decisions.

**Note:** The choice to put threads within a process does not constrain implementations to implement threads in that manner. However, all functions have to behave as though threads share the indicated state information with the process from which they were created.

Threads need to share resources in order to cooperate. Memory has to be widely shared between threads in order for the threads to cooperate at a fine level of granularity. Threads keep data structures and the locks protecting those data structures in shared memory. For a data structure to be usefully shared between threads, such structures should not refer to any data that can only be interpreted meaningfully by a single thread. Thus, any system resources that might be referred to in data structures need to be shared between all threads. File descriptors, pathnames, and pointers to stack variables are all things that programmers want to share between their threads. Thus, the file descriptor table, the root directory, the current working directory, and the address space have to be shared.

Library implementations are possible as long as the effective behavior is as if system services invoked by one thread do not suspend other threads. This may be difficult for some library implementations on systems that do not provide asynchronous facilities.

See Section B.2.9 (on page 152) for additional rationale.

**Thread ID**

See Section B.2.9.2 (on page 168) for additional rationale.

**Thread-Safe Function**

All functions required by IEEE Std 1003.1-2001 need to be thread-safe; see Section A.4.16 (on page 42) and Section B.2.9.1 (on page 165) for additional rationale.

**User Database**

There are no references in IEEE Std 1003.1-2001 to a “passwd file” or a “group file”, and there is no requirement that the *group* or *passwd* databases be kept in files containing editable text. Many large timesharing systems use *passwd* databases that are hashed for speed. Certain security classifications prohibit certain information in the *passwd* database from being publicly readable.

The term “encoded” is used instead of “encrypted” in order to avoid the implementation connotations (such as reversibility or use of a particular algorithm) of the latter term.

The *getgrent()*, *setgrent()*, *endgrent()*, *getpwent()*, *setpwent()*, and *endpwent()* functions are not included as part of the base standard because they provide a linear database search capability that is not generally useful (the *getpwuid()*, *getpwnam()*, *getgrgid()*, and *getgrnam()* functions are provided for keyed lookup) and because in certain distributed systems, especially those with different authentication domains, it may not be possible or desirable to provide an application with the ability to browse the system databases indiscriminately. They are provided on XSI-conformant systems due to their historical usage by many existing applications.

A change from historical implementations is that the structures used by these functions have fields of the types **gid\_t** and **uid\_t**, which are required to be defined in the **<sys/types.h>** header. IEEE Std 1003.1-2001 requires implementations to ensure that these types are defined by inclusion of **<grp.h>** and **<pwd.h>**, respectively, without imposing any name space pollution or

1282 errors from redefinition of types.

1283 IEEE Std 1003.1-2001 is silent about the content of the strings containing user or group names.  
 1284 These could be digit strings. IEEE Std 1003.1-2001 is also silent as to whether such digit strings  
 1285 bear any relationship to the corresponding (numeric) user or group ID.

#### 1286 *Database Access*

1287 The thread-safe versions of the user and group database access functions return values in user-  
 1288 supplied buffers instead of possibly using static data areas that may be overwritten by each call.

1289 IEEE Std 1003.1-2001/Cor 2-2004, item XBD/TC2/D6/11 is applied, removing the words “of 2  
 1290 implementation-defined format”. 2

#### 1291 **Virtual Processor\***

1292 The term “virtual processor” was chosen as a neutral term describing all kernel-level  
 1293 schedulable entities, such as processes, Mach tasks, or lightweight processes. Implementing  
 1294 threads using multiple processes as virtual processors, or implementing multiplexed threads  
 1295 above a virtual processor layer, should be possible, provided some mechanism has also been  
 1296 implemented for sharing state between processes or virtual processors. Many systems may also  
 1297 wish to provide implementations of threads on systems providing “shared processes” or  
 1298 “variable-weight processes”. It was felt that exposing such implementation details would  
 1299 severely limit the type of systems upon which the threads interface could be supported and  
 1300 prevent certain types of valid implementations. It was also determined that a virtual processor  
 1301 interface was out of the scope of the Rationale (Informative) volume of IEEE Std 1003.1-2001.

#### 1302 **XSI**

1303 This is introduced to allow IEEE Std 1003.1-2001 to be adopted as an IEEE standard and an Open  
 1304 Group Technical Standard, serving both the POSIX and the Single UNIX Specification in a core  
 1305 set of volumes.

1306 The term “XSI” has been used for 10 years in connection with the XPG series and the first and  
 1307 second versions of the base volumes of the Single UNIX Specification. The XSI margin code was  
 1308 introduced to denote the extended or more restrictive semantics beyond POSIX that are  
 1309 applicable to UNIX systems.

## 1310 **A.4 General Concepts**

### 1311 **A.4.1 Concurrent Execution**

1312 There is no additional rationale provided for this section.

### 1313 **A.4.2 Directory Protection**

1314 There is no additional rationale provided for this section.

### 1315 A.4.3 Extended Security Controls

1316 Allowing an implementation to define extended security controls enables the use of  
 1317 IEEE Std 1003.1-2001 in environments that require different or more rigorous security than that  
 1318 provided in POSIX.1. Extensions are allowed in two areas: privilege and file access permissions.  
 1319 The semantics of these areas have been defined to permit extensions with reasonable, but not  
 1320 exact, compatibility with all existing practices. For example, the elimination of the superuser  
 1321 definition precludes identifying a process as privileged or not by virtue of its effective user ID.

### 1322 A.4.4 File Access Permissions

1323 A process should not try to anticipate the result of an attempt to access data by *a priori* use of  
 1324 these rules. Rather, it should make the attempt to access data and examine the return value (and  
 1325 possibly *errno* as well), or use *access()*. An implementation may include other security  
 1326 mechanisms in addition to those specified in POSIX.1, and an access attempt may fail because of  
 1327 those additional mechanisms, even though it would succeed according to the rules given in this  
 1328 section. (For example, the user's security level might be lower than that of the object of the access  
 1329 attempt.) The supplementary group IDs provide another reason for a process to not attempt to  
 1330 anticipate the result of an access attempt.

### 1331 A.4.5 File Hierarchy

1332 Though the file hierarchy is commonly regarded to be a tree, POSIX.1 does not define it as such  
 1333 for three reasons:

- 1334 1. Links may join branches.
- 1335 2. In some network implementations, there may be no single absolute root directory; see  
 1336 *pathname resolution*.
- 1337 3. With symbolic links, the file system need not be a tree or even a directed acyclic graph.

### 1338 A.4.6 Filenames

1339 Historically, certain filenames have been reserved. This list includes **core**, **/etc/passwd**, and so  
 1340 on. Conforming applications should avoid these.

1341 Most historical implementations prohibit case folding in filenames; that is, treating uppercase  
 1342 and lowercase alphabetic characters as identical. However, some consider case folding desirable:

- 1343 • For user convenience
- 1344 • For ease-of-implementation of the POSIX.1 interface as a hosted system on some popular  
 1345 operating systems

1346 Variants, such as maintaining case distinctions in filenames, but ignoring them in comparisons,  
 1347 have been suggested. Methods of allowing escaped characters of the case opposite the default  
 1348 have been proposed.

1349 Many reasons have been expressed for not allowing case folding, including:

- 1350 • No solid evidence has been produced as to whether case-sensitivity or case-insensitivity is  
 1351 more convenient for users.
- 1352 • Making case-insensitivity a POSIX.1 implementation option would be worse than either  
 1353 having it or not having it, because:
  - 1354 — More confusion would be caused among users.

- 1355 — Application developers would have to account for both cases in their code.
- 1356 — POSIX.1 implementors would still have other problems with native file systems, such as
- 1357 short or otherwise constrained filenames or pathnames, and the lack of hierarchical
- 1358 directory structure.
- 1359 • Case folding is not easily defined in many European languages, both because many of them
- 1360 use characters outside the US ASCII alphabetic set, and because:
- 1361 — In Spanish, the digraph "ll" is considered to be a single letter, the capitalized form of
- 1362 which may be either "Ll" or "LL", depending on context.
- 1363 — In French, the capitalized form of a letter with an accent may or may not retain the accent,
- 1364 depending on the country in which it is written.
- 1365 — In German, the sharp ess may be represented as a single character resembling a Greek
- 1366 beta (ß) in lowercase, but as the digraph "SS" in uppercase.
- 1367 — In Greek, there are several lowercase forms of some letters; the one to use depends on its
- 1368 position in the word. Arabic has similar rules.
- 1369 • Many East Asian languages, including Japanese, Chinese, and Korean, do not distinguish
- 1370 case and are sometimes encoded in character sets that use more than one byte per character.
- 1371 • Multiple character codes may be used on the same machine simultaneously. There are
- 1372 several ISO character sets for European alphabets. In Japan, several Japanese character codes
- 1373 are commonly used together, sometimes even in filenames; this is evidently also the case in
- 1374 China. To handle case insensitivity, the kernel would have to at least be able to distinguish
- 1375 for which character sets the concept made sense.
- 1376 • The file system implementation historically deals only with bytes, not with characters, except
- 1377 for slash and the null byte.
- 1378 • The purpose of POSIX.1 is to standardize the common, existing definition, not to change it.
- 1379 Mandating case-insensitivity would make all historical implementations non-standard.
- 1380 • Not only the interface, but also application programs would need to change, counter to the
- 1381 purpose of having minimal changes to existing application code.
- 1382 • At least one of the original developers of the UNIX system has expressed objection in the
- 1383 strongest terms to either requiring case-insensitivity or making it an option, mostly on the
- 1384 basis that POSIX.1 should not hinder portability of application programs across related
- 1385 implementations in order to allow compatibility with unrelated operating systems.
- 1386 Two proposals were entertained regarding case folding in filenames:
- 1387 1. Remove all wording that previously permitted case folding.
- 1388 Rationale Case folding is inconsistent with portable filename character set definition
- 1389 and filename definition (all characters except slash and null). No known
- 1390 implementations allowing all characters except slash and null also do case
- 1391 folding.
- 1392 2. Change “though this practice is not recommended:” to “although this practice is strongly
- 1393 discouraged.”
- 1394 Rationale If case folding must be included in POSIX.1, the wording should be stronger
- 1395 to discourage the practice.
- 1396 The consensus selected the first proposal. Otherwise, a conforming application would have to
- 1397 assume that case folding would occur when it was not wanted, but that it would not occur when



1398 it was wanted.

#### 1399 **A.4.7 File Times Update**

1400 This section reflects the actions of historical implementations. The times are not updated  
1401 immediately, but are only marked for update by the functions. An implementation may update  
1402 these times immediately.

1403 The accuracy of the time update values is intentionally left unspecified so that systems can  
1404 control the bandwidth of a possible covert channel.

1405 The wording was carefully chosen to make it clear that there is no requirement that the  
1406 conformance document contain information that might incidentally affect file update times. Any  
1407 function that performs pathname resolution might update several *st\_atime* fields. Functions such  
1408 as *getpwnam()* and *getgrnam()* might update the *st\_atime* field of some specific file or files. It is  
1409 intended that these are not required to be documented in the conformance document, but they  
1410 should appear in the system documentation.

#### 1411 **A.4.8 Host and Network Byte Order**

1412 There is no additional rationale provided for this section.

#### 1413 **A.4.9 Measurement of Execution Time**

1414 The methods used to measure the execution time of processes and threads, and the precision of  
1415 these measurements, may vary considerably depending on the software architecture of the  
1416 implementation, and on the underlying hardware. Implementations can also make tradeoffs  
1417 between the scheduling overhead and the precision of the execution time measurements.  
1418 IEEE Std 1003.1-2001 does not impose any requirement on the accuracy of the execution time; it  
1419 instead specifies that the measurement mechanism and its precision are implementation-  
1420 defined.

#### 1421 **A.4.10 Memory Synchronization**

1422 In older multi-processors, access to memory by the processors was strictly multiplexed. This  
1423 meant that a processor executing program code interrogates or modifies memory in the order  
1424 specified by the code and that all the memory operation of all the processors in the system  
1425 appear to happen in some global order, though the operation histories of different processors are  
1426 interleaved arbitrarily. The memory operations of such machines are said to be sequentially  
1427 consistent. In this environment, threads can synchronize using ordinary memory operations. For  
1428 example, a producer thread and a consumer thread can synchronize access to a circular data  
1429 buffer as follows:

```

1430     int rdptr = 0;
1431     int wrptr = 0;
1432     data_t buf[BUFSIZE];

1433     Thread 1:
1434         while (work_to_do) {
1435             int next;

1436             buf[wrptr] = produce();
1437             next = (wrptr + 1) % BUFSIZE;
1438             while (rdptr == next)
1439                 ;
1440             wrptr = next;
1441         }

1442     Thread 2:
1443         while (work_to_do) {
1444             while (rdptr == wrptr)
1445                 ;
1446             consume(buf[rdptr]);
1447             rdptr = (rdptr + 1) % BUFSIZE;
1448         }

```

In modern multi-processors, these conditions are relaxed to achieve greater performance. If one processor stores values in location A and then location B, then other processors loading data from location B and then location A may see the new value of B but the old value of A. The memory operations of such machines are said to be weakly ordered. On these machines, the circular buffer technique shown in the example will fail because the consumer may see the new value of *wrptr* but the old value of the data in the buffer. In such machines, synchronization can only be achieved through the use of special instructions that enforce an order on memory operations. Most high-level language compilers only generate ordinary memory operations to take advantage of the increased performance. They usually cannot determine when memory operation order is important and generate the special ordering instructions. Instead, they rely on the programmer to use synchronization primitives correctly to ensure that modifications to a location in memory are ordered with respect to modifications and/or access to the same location in other threads. Access to read-only data need not be synchronized. The resulting program is said to be data race-free.

Synchronization is still important even when accessing a single primitive variable (for example, an integer). On machines where the integer may not be aligned to the bus data width or be larger than the data width, a single memory load may require multiple memory cycles. This means that it may be possible for some parts of the integer to have an old value while other parts have a newer value. On some processor architectures this cannot happen, but portable programs cannot rely on this.

In summary, a portable multi-threaded program, or a multi-process program that shares writable memory between processes, has to use the synchronization primitives to synchronize data access. It cannot rely on modifications to memory being observed by other threads in the order written in the application or even on modification of a single variable being seen atomically.

Conforming applications may only use the functions listed to synchronize threads of control with respect to memory access. There are many other candidates for functions that might also be used. Examples are: signal sending and reception, or pipe writing and reading. In general, any function that allows one thread of control to wait for an action caused by another thread of control is a candidate. IEEE Std 1003.1-2001 does not require these additional functions to

synchronize memory access since this would imply the following:

- All these functions would have to be recognized by advanced compilation systems so that memory operations and calls to these functions are not reordered by optimization.
- All these functions would potentially have to have memory synchronization instructions added, depending on the particular machine.
- The additional functions complicate the model of how memory is synchronized and make automatic data race detection techniques impractical.

Formal definitions of the memory model were rejected as unreadable by the vast majority of programmers. In addition, most of the formal work in the literature has concentrated on the memory as provided by the hardware as opposed to the application programmer through the compiler and runtime system. It was believed that a simple statement intuitive to most programmers would be most effective. IEEE Std 1003.1-2001 defines functions that can be used to synchronize access to memory, but it leaves open exactly how one relates those functions to the semantics of each function as specified elsewhere in IEEE Std 1003.1-2001. IEEE Std 1003.1-2001 also does not make a formal specification of the partial ordering in time that the functions can impose, as that is implied in the description of the semantics of each function. It simply states that the programmer has to ensure that modifications do not occur “simultaneously” with other access to a memory location.

IEEE Std 1003.1-2001/Cor 1-2002, item XBD/TC1/D6/4 is applied, adding a new paragraph beneath the table of functions: “The *pthread\_once()* function shall synchronize memory for the first call in each thread for a given *pthread\_once\_t* object.”.

#### A.4.11 Pathname Resolution

It is necessary to differentiate between the definition of pathname and the concept of pathname resolution with respect to the handling of trailing slashes. By specifying the behavior here, it is not possible to provide an implementation that is conforming but extends all interfaces that handle pathnames to also handle strings that are not legal pathnames (because they have trailing slashes).

Pathnames that end with one or more trailing slash characters must refer to directory paths. Previous versions of IEEE Std 1003.1-2001 were not specific about the distinction between trailing slashes on files and directories, and both were permitted.

Two types of implementation have been prevalent; those that ignored trailing slash characters on all pathnames regardless, and those that permitted them only on existing directories.

IEEE Std 1003.1-2001 requires that a pathname with a trailing slash character be treated as if it had a trailing “/.” everywhere.

Note that this change does not break any conforming applications; since there were two different types of implementation, no application could have portably depended on either behavior. This change does however require some implementations to be altered to remain compliant. Substantial discussion over a three-year period has shown that the benefits to application developers outweighs the disadvantages for some vendors.

On a historical note, some early applications automatically appended a “/” to every path. Rather than fix the applications, the system implementation was modified to accept this behavior by ignoring any trailing slash.

Each directory has exactly one parent directory which is represented by the name **dot-dot** in the first directory. No other directory, regardless of linkages established by symbolic links, is considered the parent directory by IEEE Std 1003.1-2001.

There are two general categories of interfaces involving pathname resolution: those that follow the symbolic link, and those that do not. There are several exceptions to this rule; for example, `open(path, O_CREAT|O_EXCL)` will fail when `path` names a symbolic link. However, in all other situations, the `open()` function will follow the link.

What the filename **dot-dot** refers to relative to the root directory is implementation-defined. In Version 7 it refers to the root directory itself; this is the behavior mentioned in IEEE Std 1003.1-2001. In some networked systems the construction `././hostname/` is used to refer to the root directory of another host, and POSIX.1 permits this behavior.

Other networked systems use the construct `//hostname` for the same purpose; that is, a double initial slash is used. There is a potential problem with existing applications that create full pathnames by taking a trunk and a relative pathname and making them into a single string separated by `'/'`, because they can accidentally create networked pathnames when the trunk is `'/'`. This practice is not prohibited because such applications can be made to conform by simply changing to use `"/"` as a separator instead of `'/'`:

- If the trunk is `'/'`, the full pathname will begin with `"/"` (the initial `'/'` and the separator `"/"`). This is the same as `'/'`, which is what is desired. (This is the general case of making a relative pathname into an absolute one by prefixing with `"/"` instead of `'/'`.)
- If the trunk is `"/A"`, the result is `"/A/. . ."`; since non-leading sequences of two or more slashes are treated as a single slash, this is equivalent to the desired `"/A/. . ."`.
- If the trunk is `"//A"`, the implementation-defined semantics will apply. (The multiple slash rule would apply.)

Application developers should avoid generating pathnames that start with `"/"`. Implementations are strongly encouraged to avoid using this special interpretation since a number of applications currently do not follow this practice and may inadvertently generate `"/. . ."`.

The term “root directory” is only defined in POSIX.1 relative to the process. In some implementations, there may be no absolute root directory. The initialization of the root directory of a process is implementation-defined.

#### A.4.12 Process ID Reuse

There is no additional rationale provided for this section.

#### A.4.13 Scheduling Policy

There is no additional rationale provided for this section.

#### A.4.14 Seconds Since the Epoch

Coordinated Universal Time (UTC) includes leap seconds. However, in POSIX time (seconds since the Epoch), leap seconds are ignored (not applied) to provide an easy and compatible method of computing time differences. Broken-down POSIX time is therefore not necessarily UTC, despite its appearance.

As of September 2000, 24 leap seconds had been added to UTC since the Epoch, 1 January, 1970. Historically, one leap second is added every 15 months on average, so this offset can be expected to grow steadily with time.

Most systems' notion of “time” is that of a continuously increasing value, so this value should increase even during leap seconds. However, not only do most systems not keep track of leap seconds, but most systems are probably not synchronized to any standard time reference.

Therefore, it is inappropriate to require that a time represented as seconds since the Epoch precisely represent the number of seconds between the referenced time and the Epoch.

It is sufficient to require that applications be allowed to treat this time as if it represented the number of seconds between the referenced time and the Epoch. It is the responsibility of the vendor of the system, and the administrator of the system, to ensure that this value represents the number of seconds between the referenced time and the Epoch as closely as necessary for the application being run on that system.

It is important that the interpretation of time names and seconds since the Epoch values be consistent across conforming systems; that is, it is important that all conforming systems interpret “536 457 599 seconds since the Epoch” as 59 seconds, 59 minutes, 23 hours 31 December 1986, regardless of the accuracy of the system’s idea of the current time. The expression is given to ensure a consistent interpretation, not to attempt to specify the calendar. The relationship between *tm\_yday* and the day of week, day of month, and month is in accordance with the Gregorian calendar, and so is not specified in POSIX.1.

Consistent interpretation of seconds since the Epoch can be critical to certain types of distributed applications that rely on such timestamps to synchronize events. The accrual of leap seconds in a time standard is not predictable. The number of leap seconds since the Epoch will likely increase. POSIX.1 is more concerned about the synchronization of time between applications of astronomically short duration.

Note that *tm\_yday* is zero-based, not one-based, so the day number in the example above is 364. Note also that the division is an integer division (discarding remainder) as in the C language.

Note also that the meaning of *gmtime()*, *localtime()*, and *mktime()* is specified in terms of this expression. However, the ISO C standard computes *tm\_yday* from *tm\_mday*, *tm\_mon*, and *tm\_year* in *mktime()*. Because it is stated as a (bidirectional) relationship, not a function, and because the conversion between month-day-year and day-of-year dates is presumed well known and is also a relationship, this is not a problem.

Implementations that implement **time\_t** as a signed 32-bit integer will overflow in 2038. The data size for **time\_t** is as per the ISO C standard definition, which is implementation-defined.

See also **Epoch** (on page 17).

The topic of whether seconds since the Epoch should account for leap seconds has been debated on a number of occasions, and each time consensus was reached (with acknowledged dissent each time) that the majority of users are best served by treating all days identically. (That is, the majority of applications were judged to assume a single length—as measured in seconds since the Epoch—for all days. Thus, leap seconds are not applied to seconds since the Epoch.) Those applications which do care about leap seconds can determine how to handle them in whatever way those applications feel is best. This was particularly emphasized because there was disagreement about what the best way of handling leap seconds might be. It is a practical impossibility to mandate that a conforming implementation must have a fixed relationship to any particular official clock (consider isolated systems, or systems performing “reruns” by setting the clock to some arbitrary time).

Note that as a practical consequence of this, the length of a second as measured by some external standard is not specified. This unspecified second is nominally equal to an International System (SI) second in duration. Applications must be matched to a system that provides the particular handling of external time in the way required by the application.

IEEE Std 1003.1-2001/Cor 2-2004, item XBD/TC2/D6/12 is applied, making an editorial correction to the paragraph commencing “How any changes to the value of seconds ...”. 2

1613 **A.4.15 Semaphore**

1614 There is no additional rationale provided for this section.

1615 **A.4.16 Thread-Safety**

1616 Where the interface of a function required by IEEE Std 1003.1-2001 precludes thread-safety, an  
 1617 alternate thread-safe form is provided. The names of these thread-safe forms are the same as the  
 1618 non-thread-safe forms with the addition of the suffix “\_r”. The suffix “\_r” is historical, where  
 1619 the ‘*r*’ stood for “reentrant”.

1620 In some cases, thread-safety is provided by restricting the arguments to an existing function.

1621 See also Section B.2.9.1 (on page 165).

1622 **A.4.17 Tracing**

1623 Refer to Section B.2.11 (on page 181).

1624 **A.4.18 Treatment of Error Conditions for Mathematical Functions**

1625 There is no additional rationale provided for this section.

1626 **A.4.19 Treatment of NaN Arguments for Mathematical Functions**

1627 There is no additional rationale provided for this section.

1628 **A.4.20 Utility**

1629 There is no additional rationale provided for this section.

1630 **A.4.21 Variable Assignment**

1631 There is no additional rationale provided for this section.

1632 **A.5 File Format Notation**

1633 The notation for spaces allows some flexibility for application output. Note that an empty  
 1634 character position in *format* represents one or more <blank>s on the output (not *white space*,  
 1635 which can include <newline>s). Therefore, another utility that reads that output as its input  
 1636 must be prepared to parse the data using *scanf()*, *awk*, and so on. The ‘*Δ*’ character is used when  
 1637 exactly one <space> is output.

1638 The treatment of integers and spaces is different from the *printf()* function in that they can be  
 1639 surrounded with <blank>s. This was done so that, given a format such as:

1640 `"%d\n", <foo>`1641 the implementation could use a *printf()* call such as:1642 `printf("%6d\n", foo);`1643 and still conform. This notation is thus somewhat like *scanf()* in addition to *printf()*.

1644 The *printf()* function was chosen as a model because most of the standard developers were  
 1645 familiar with it. One difference from the C function *printf()* is that the *l* and *h* conversion  
 1646 specifier characters are not used. As expressed by the Shell and Utilities volume of  
 1647 IEEE Std 1003.1-2001, there is no differentiation between decimal values for type **int**, type **long**,

1648 or type **short**. The conversion specifications %d or %i should be interpreted as an arbitrary  
 1649 length sequence of digits. Also, no distinction is made between single precision and double  
 1650 precision numbers (**float** or **double** in C). These are simply referred to as floating-point numbers.

1651 Many of the output descriptions in the Shell and Utilities volume of IEEE Std 1003.1-2001 use the  
 1652 term “line”, such as:

1653       "%s", <input line>

1654 Since the definition of *line* includes the trailing <newline> already, there is no need to include a  
 1655 ‘\n’ in the format; a double <newline> would otherwise result.

## 1656 **A.6 Character Set**

### 1657 **A.6.1 Portable Character Set**

1658 The portable character set is listed in full so there is no dependency on the ISO/IEC 646:1991  
 1659 standard (or historically ASCII) encoded character set, although the set is identical to the  
 1660 characters defined in the International Reference version of the ISO/IEC 646:1991 standard.

1661 IEEE Std 1003.1-2001 poses no requirement that multiple character sets or codesets be supported,  
 1662 leaving this as a marketing differentiation for implementors. Although multiple charmap files  
 1663 are supported, it is the responsibility of the implementation to provide the file(s); if only one is  
 1664 provided, only that one will be accessible using the *localedef -f* option.

1665 The statement about invariance in codesets for the portable character set is worded to avoid  
 1666 precluding implementations where multiple incompatible codesets are available (for instance,  
 1667 ASCII and EBCDIC). The standard utilities cannot be expected to produce predictable results if  
 1668 they access portable characters that vary on the same implementation.

1669 Not all character sets need include the portable character set, but each locale must include it. For  
 1670 example, a Japanese-based locale might be supported by a mixture of character sets: JIS X 0201  
 1671 Roman (a Japanese version of the ISO/IEC 646:1991 standard), JIS X 0208, and JIS X 0201  
 1672 Katakana. Not all of these character sets include the portable characters, but at least one does  
 1673 (JIS X 0201 Roman).

### 1674 **A.6.2 Character Encoding**

1675 Encoding mechanisms based on single shifts, such as the EUC encoding used in some Asian and  
 1676 other countries, can be supported via the current charmap mechanism. With single-shift  
 1677 encoding, each character is preceded by a shift code (SS2 or SS3). A complete EUC code,  
 1678 consisting of the portable character set (G0) and up to three additional character sets (G1, G2,  
 1679 G3), can be described using the current charmap mechanism; the encoding for each character in  
 1680 additional character sets G2 and G3 must then include their single-shift code. Other mechanisms  
 1681 to support locales based on encoding mechanisms such as locking shift are not addressed by this  
 1682 volume of IEEE Std 1003.1-2001.

### 1683 A.6.3 C Language Wide-Character Codes

1684 The standard does not specify how wide characters are encoded or provide a method for 2  
 1685 defining wide characters in a charmap. It specifies ways of translating between wide characters 2  
 1686 and multi-byte characters. The standard does not prevent an extension from providing a method 2  
 1687 to define wide characters. 2

1688 IEEE Std 1003.1-2001/Cor 2-2004, item XBD/TC2/D6/13 is applied, adding a statement that the 2  
 1689 standard has no means of defining a wide-character codeset. 2

### 1690 A.6.4 Character Set Description File

1691 IEEE PASC Interpretation 1003.2 #196 is applied, removing three lines of text dealing with  
 1692 ranges of symbolic names using position constant values which had been erroneously included  
 1693 in the final IEEE P1003.2b draft standard.

1694 IEEE Std 1003.1-2001/Cor 2-2004, item XBD/TC2/D6/14 is applied, correcting the example and 2  
 1695 adding a statement that the standard provides no means of defining a wide-character codeset. 2

1696 IEEE Std 1003.1-2001/Cor 2-2004, item XBD/TC2/D6/15 is applied, allowing the value zero for 2  
 1697 the width value of **WIDTH** and **WIDTH\_DEFAULT**. This is required to cover some existing 2  
 1698 locales. 2

#### 1699 A.6.4.1 State-Dependent Character Encodings

1700 A requirement was considered that would force utilities to eliminate any redundant locking  
 1701 shifts, but this was left as a quality of implementation issue.

1702 This change satisfies the following requirement from the ISO POSIX-2:1993 standard, Annex  
 1703 H.1:

1704 *The support of state-dependent (shift encoding) character sets should be addressed fully. See*  
 1705 *descriptions of these in the Base Definitions volume of IEEE Std 1003.1-2001, Section 6.2, Character*  
 1706 *Encoding. If such character encodings are supported, it is expected that this will impact the Base*  
 1707 *Definitions volume of IEEE Std 1003.1-2001, Section 6.2, Character Encoding, the Base Definitions*  
 1708 *volume of IEEE Std 1003.1-2001, Chapter 7, Locale, the Base Definitions volume of*  
 1709 *IEEE Std 1003.1-2001, Chapter 9, Regular Expressions, and the comm, cut, diff, grep, head, join,*  
 1710 *paste, and tail utilities.*

1711 The character set description file provides:

- 1712 • The capability to describe character set attributes (such as collation order or character  
 1713 classes) independent of character set encoding, and using only the characters in the portable  
 1714 character set. This makes it possible to create generic *localedef* source files for all codesets that  
 1715 share the portable character set (such as the ISO 8859 family or IBM Extended ASCII).
- 1716 • Standardized symbolic names for all characters in the portable character set, making it  
 1717 possible to refer to any such character regardless of encoding.

1718 Implementations are free to choose their own symbolic names, as long as the names identified  
 1719 by the Base Definitions volume of IEEE Std 1003.1-2001 are also defined; this provides support  
 1720 for already existing “character names”.

1721 The names selected for the members of the portable character set follow the  
 1722 ISO/IEC 8859-1:1998 standard and the ISO/IEC 10646-1:2000 standard. However, several  
 1723 commonly used UNIX system names occur as synonyms in the list:

- 1724 • The historical UNIX system names are used for control characters.



- 1725 • The word “slash” is given in addition to “solidus”.
  - 1726 • The word “backslash” is given in addition to “reverse-solidus”.
  - 1727 • The word “hyphen” is given in addition to “hyphen-minus”.
  - 1728 • The word “period” is given in addition to “full-stop”.
  - 1729 • For digits, the word “digit” is eliminated.
  - 1730 • For letters, the words “Latin Capital Letter” and “Latin Small Letter” are eliminated.
  - 1731 • The words “left brace” and “right brace” are given in addition to “left-curly-bracket” and
  - 1732 “right-curly-bracket”.
  - 1733 • The names of the digits are preferred over the numbers to avoid possible confusion between
  - 1734 ‘0’ and ‘o’, and between ‘1’ and ‘l’ (one and the letter ell).
- 1735 The names for the control characters in the Base Definitions volume of IEEE Std 1003.1-2001,
- 1736 Chapter 6, Character Set were taken from the ISO/IEC 4873: 1991 standard.
- 1737 The charmap file was introduced to resolve problems with the portability of, especially, *localedef*
- 1738 sources. IEEE Std 1003.1-2001 assumes that the portable character set is constant across all
- 1739 locales, but does not prohibit implementations from supporting two incompatible codings, such
- 1740 as both ASCII and EBCDIC. Such dual-support implementations should have all charmaps and
- 1741 *localedef* sources encoded using one portable character set, in effect cross-compiling for the other
- 1742 environment. Naturally, charmaps (and *localedef* sources) are only portable without
- 1743 transformation between systems using the same encodings for the portable character set. They
- 1744 can, however, be transformed between two sets using only a subset of the actual characters (the
- 1745 portable character set). However, the particular coded character set used for an application or an
- 1746 implementation does not necessarily imply different characteristics or collation; on the contrary,
- 1747 these attributes should in many cases be identical, regardless of codeset. The charmap provides
- 1748 the capability to define a common locale definition for multiple codesets (the same *localedef*
- 1749 source can be used for codesets with different extended characters; the ability in the charmap to
- 1750 define empty names allows for characters missing in certain codesets).
- 1751 The `<escape_char>` declaration was added at the request of the international community to ease
- 1752 the creation of portable charmap files on terminals not implementing the default backslash
- 1753 escape. The `<comment_char>` declaration was added at the request of the international
- 1754 community to eliminate the potential confusion between the number sign and the pound sign.
- 1755 The octal number notation with no leading zero required was selected to match those of *awk* and
- 1756 *tr* and is consistent with that used by *localedef*. To avoid confusion between an octal constant
- 1757 and the back-references used in *localedef* source, the octal, hexadecimal, and decimal constants
- 1758 must contain at least two digits. As single-digit constants are relatively rare, this should not
- 1759 impose any significant hardship. Provision is made for more digits to account for systems in
- 1760 which the byte size is larger than 8 bits. For example, a Unicode (ISO/IEC 10646-1:2000
- 1761 standard) system that has defined 16-bit bytes may require six octal, four hexadecimal, and five
- 1762 decimal digits.
- 1763 The decimal notation is supported because some newer international standards define character
- 1764 values in decimal, rather than in the old column/row notation.
- 1765 The charmap identifies the coded character sets supported by an implementation. At least one
- 1766 charmap must be provided, but no implementation is required to provide more than one.
- 1767 Likewise, implementations can allow users to generate new charmaps (for instance, for a new
- 1768 version of the ISO 8859 family of coded character sets), but does not have to do so. If users are
- 1769 allowed to create new charmaps, the system documentation describes the rules that apply (for
- 1770 instance, “only coded character sets that are supersets of the ISO/IEC 646: 1991 standard IRV, no

1771 multi-byte characters”).

1772 This addition of the **WIDTH** specification satisfies the following requirement from the  
1773 ISO POSIX-2: 1993 standard, Annex H.1:

1774 (9) *The definition of column position relies on the implementation’s knowledge of the integral width*  
1775 *of the characters. The charmap or LC\_CTYPE locale definitions should be enhanced to allow*  
1776 *application specification of these widths.*

1777 The character “width” information was first considered for inclusion under *LC\_CTYPE* but was  
1778 moved because it is more closely associated with the information in the charmap than  
1779 information in the locale source (cultural conventions information). Concerns were raised that  
1780 formalizing this type of information is moving the locale source definition from the codeset-  
1781 independent entity that it was designed to be to a repository of codeset-specific information. A  
1782 similar issue occurred with the `<code_set_name>`, `<mb_cur_max>`, and `<mb_cur_min>`  
1783 information, which was resolved to reside in the charmap definition.

1784 The width definition was added to the IEEE P1003.2b draft standard with the intent that the  
1785 `wcswidth()` and/or `wcwidth()` functions (currently specified in the System Interfaces volume of  
1786 IEEE Std 1003.1-2001) be the mechanism to retrieve the character width information.

## 1787 A.7 Locale

### 1788 A.7.1 General

1789 The description of locales is based on work performed in the UniForum Technical Committee,  
1790 Subcommittee on Internationalization. Wherever appropriate, keywords are taken from the  
1791 ISO C standard or the X/Open Portability Guide.

1792 The value used to specify a locale with environment variables is the name specified as the *name*  
1793 operand to the *localedef* utility when the locale was created. This provides a verifiable method to  
1794 create and invoke a locale.

1795 The “object” definitions need not be portable, as long as “source” definitions are. Strictly  
1796 speaking, source definitions are portable only between implementations using the same  
1797 character set(s). Such source definitions, if they use symbolic names only, easily can be ported  
1798 between systems using different codesets, as long as the characters in the portable character set  
1799 (see the Base Definitions volume of IEEE Std 1003.1-2001, Section 6.1, Portable Character Set)  
1800 have common values between the codesets; this is frequently the case in historical  
1801 implementations. Of source, this requires that the symbolic names used for characters outside  
1802 the portable character set be identical between character sets. The definition of symbolic names  
1803 for characters is outside the scope of IEEE Std 1003.1-2001, but is certainly within the scope of  
1804 other standards organizations.

1805 Applications can select the desired locale by invoking the *setlocale()* function (or equivalent)  
1806 with the appropriate value. If the function is invoked with an empty string, the value of the  
1807 corresponding environment variable is used. If the environment variable is not set or is set to the  
1808 empty string, the implementation sets the appropriate environment as defined in the Base  
1809 Definitions volume of IEEE Std 1003.1-2001, Chapter 8, Environment Variables.

**1810 A.7.2 POSIX Locale**

1811 The POSIX locale is equal to the C locale. To avoid being classified as a C-language function, the  
1812 name has been changed to the POSIX locale; the environment variable value can be either  
1813 "POSIX" or, for historical reasons, "C".

1814 The POSIX definitions mirror the historical UNIX system behavior.

1815 The use of symbolic names for characters in the tables does not imply that the POSIX locale must  
1816 be described using symbolic character names, but merely that it may be advantageous to do so.

**1817 A.7.3 Locale Definition**

1818 The decision to separate the file format from the *localedef* utility description was only partially  
1819 editorial. Implementations may provide other interfaces than *localedef*. Requirements on “the  
1820 utility”, mostly concerning error messages, are described in this way because they are meant to  
1821 affect the other interfaces implementations may provide as well as *localedef*.

1822 The text about POSIX2\_LOCALEDEF does not mean that internationalization is optional; only  
1823 that the functionality of the *localedef* utility is. REs, for instance, must still be able to recognize,  
1824 for example, character class expressions such as "[[:alpha:]]". A possible analogy is with  
1825 an applications development environment; while all conforming implementations must be  
1826 capable of executing applications, not all need to have the development environment installed.  
1827 The assumption is that the capability to modify the behavior of utilities (and applications) via  
1828 locale settings must be supported. If the *localedef* utility is not present, then the only choice is to  
1829 select an existing (presumably implementation-documented) locale. An implementation could,  
1830 for example, choose to support only the POSIX locale, which would in effect limit the amount of  
1831 changes from historical implementations quite drastically. The *localedef* utility is still required,  
1832 but would always terminate with an exit code indicating that no locale could be created.  
1833 Supported locales must be documented using the syntax defined in this chapter. (This ensures  
1834 that users can accurately determine what capabilities are provided. If the implementation  
1835 decides to provide additional capabilities to the ones in this chapter, that is already provided  
1836 for.)

1837 If the option is present (that is, locales can be created), then the *localedef* utility must be capable  
1838 of creating locales based on the syntax and rules defined in this chapter. This does not mean that  
1839 the implementation cannot also provide alternate means for creating locales.

1840 The octal, decimal, and hexadecimal notations are the same employed by the charmap facility  
1841 (see the Base Definitions volume of IEEE Std 1003.1-2001, Section 6.4, Character Set Description  
1842 File). To avoid confusion between an octal constant and a back-reference, the octal, hexadecimal,  
1843 and decimal constants must contain at least two digits. As single-digit constants are relatively  
1844 rare, this should not impose any significant hardship. Provision is made for more digits to  
1845 account for systems in which the byte size is larger than 8 bits. For example, a Unicode (see the  
1846 ISO/IEC 10646-1:2000 standard) system that has defined 16-bit bytes may require six octal, four  
1847 hexadecimal, and five decimal digits. As with the charmap file, multi-byte characters are  
1848 described in the locale definition file using “big-endian” notation for reasons of portability.  
1849 There is no requirement that the internal representation in the computer memory be in this same  
1850 order.

1851 One of the guidelines used for the development of this volume of IEEE Std 1003.1-2001 is that  
1852 characters outside the invariant part of the ISO/IEC 646:1991 standard should not be used in  
1853 portable specifications. The backslash character is not in the invariant part; the number sign is,  
1854 but with multiple representations: as a number sign, and as a pound sign. As far as general  
1855 usage of these symbols, they are covered by the “grandfather clause”, but for newly defined  
1856 interfaces, the WG15 POSIX working group has requested that POSIX provide alternate

representations. Consequently, while the default escape character remains the backslash and the default comment character is the number sign, implementations are required to recognize alternative representations, identified in the applicable source file via the `<escape_char>` and `<comment_char>` keywords.

#### A.7.3.1 *LC\_CTYPE*

The *LC\_CTYPE* category is primarily used to define the encoding-independent aspects of a character set, such as character classification. In addition, certain encoding-dependent characteristics are also defined for an application via the *LC\_CTYPE* category. IEEE Std 1003.1-2001 does not mandate that the encoding used in the locale is the same as the one used by the application because an implementation may decide that it is advantageous to define locales in a system-wide encoding rather than having multiple, logically identical locales in different encodings, and to convert from the application encoding to the system-wide encoding on usage. Other implementations could require encoding-dependent locales.

In either case, the *LC\_CTYPE* attributes that are directly dependent on the encoding, such as `<mb_cur_max>` and the display width of characters, are not user-specifiable in a locale source and are consequently not defined as keywords.

Implementations may define additional keywords or extend the *LC\_CTYPE* mechanism to allow application-defined keywords.

The text “The ellipsis specification shall only be valid within a single encoded character set” is present because it is possible to have a locale supported by multiple character encodings, as explained in the rationale for the Base Definitions volume of IEEE Std 1003.1-2001, Section 6.1, Portable Character Set. An example given there is of a possible Japanese-based locale supported by a mixture of the character sets JIS X 0201 Roman, JIS X 0208, and JIS X 0201 Katakana. Attempting to express a range of characters across these sets is not logical and the implementation is free to reject such attempts.

As the *LC\_CTYPE* character classes are based on the ISO C standard character class definition, the category does not support multi-character elements. For instance, the German character `<sharp-s>` is traditionally classified as a lowercase letter. There is no corresponding uppercase letter; in proper capitalization of German text, the `<sharp-s>` will be replaced by “SS”; that is, by two characters. This kind of conversion is outside the scope of the **toupper** and **tolower** keywords.

Where IEEE Std 1003.1-2001 specifies that only certain characters can be specified, as for the keywords **digit** and **xdigit**, the specified characters must be from the portable character set, as shown. As an example, only the Arabic digits 0 through 9 are acceptable as digits.

The character classes **digit**, **xdigit**, **lower**, **upper**, and **space** have a set of automatically included characters. These only need to be specified if the character values (that is, encoding) differs from the implementation default values. It is not possible to define a locale without these automatically included characters unless some implementation extension is used to prevent their inclusion. Such a definition would not be a proper superset of the C locale, and thus, it might not be possible for the standard utilities to be implemented as programs conforming to the ISO C standard.

The definition of character class **digit** requires that only ten characters—the ones defining digits—can be specified; alternate digits (for example, Hindi or Kanji) cannot be specified here. However, the encoding may vary if an implementation supports more than one encoding.

The definition of character class **xdigit** requires that the characters included in character class **digit** are included here also and allows for different symbols for the hexadecimal digits 10 through 15.

1904 The inclusion of the **charclass** keyword satisfies the following requirement from the  
1905 ISO POSIX-2: 1993 standard, Annex H.1:

1906 (3) *The LC\_CTYPE (2.5.2.1) locale definition should be enhanced to allow user-specified additional*  
1907 *character classes, similar in concept to the ISO C standard Multibyte Support Extension (MSE)*  
1908 *iswctype() function.*

1909 This keyword was previously included in The Open Group specifications and is now mandated  
1910 in the Shell and Utilities volume of IEEE Std 1003.1-2001.

1911 The symbolic constant {CHARCLASS\_NAME\_MAX} was also adopted from The Open Group  
1912 specifications. Applications portability is enhanced by the use of symbolic constants.

#### 1913 A.7.3.2 LC\_COLLATE

1914 The rules governing collation depend to some extent on the use. At least five different levels of  
1915 increasingly complex collation rules can be distinguished:

- 1916 1. *Byte/machine code order*: This is the historical collation order in the UNIX system and many  
1917 proprietary operating systems. Collation is here performed character by character, without  
1918 any regard to context. The primary virtue is that it usually is quite fast and also  
1919 completely deterministic; it works well when the native machine collation sequence  
1920 matches the user expectations.
- 1921 2. *Character order*: On this level, collation is also performed character by character, without  
1922 regard to context. The order between characters is, however, not determined by the code  
1923 values, but on the expectations by the user of the “correct” order between characters. In  
1924 addition, such a (simple) collation order can specify that certain characters collate equally  
1925 (for example, uppercase and lowercase letters).
- 1926 3. *String ordering*: On this level, entire strings are compared based on relatively  
1927 straightforward rules. Several “passes” may be required to determine the order between  
1928 two strings. Characters may be ignored in some passes, but not in others; the strings may  
1929 be compared in different directions; and simple string substitutions may be performed  
1930 before strings are compared. This level is best described as “dictionary” ordering; it is  
1931 based on the spelling, not the pronunciation, or meaning, of the words.
- 1932 4. *Text search ordering*: This is a further refinement of the previous level, best described as  
1933 “telephone book ordering”; some common homonyms (words spelled differently but with  
1934 the same pronunciation) are collated together; numbers are collated as if they were spelled  
1935 out, and so on.
- 1936 5. *Semantic-level ordering*: Words and strings are collated based on their meaning; entire words  
1937 (such as “the”) are eliminated; the ordering is not deterministic. This usually requires  
1938 special software and is highly dependent on the intended use.

1939 While the historical collation order formally is at level 1, for the English language it corresponds  
1940 roughly to elements at level 2. The user expects to see the output from the *ls* utility sorted very  
1941 much as it would be in a dictionary. While telephone book ordering would be an optimal goal  
1942 for standard collation, this was ruled out as the order would be language-dependent.  
1943 Furthermore, a requirement was that the order must be determined solely from the text string  
1944 and the collation rules; no external information (for example, “pronunciation dictionaries”)   
1945 could be required.

1946 As a result, the goal for the collation support is at level 3. This also matches the requirements for  
1947 the Canadian collation order, as well as other, known collation requirements for alphabetic  
1948 scripts. It specifically rules out collation based on pronunciation rules or based on semantic

1949	analysis of the text.
1950	The syntax for the <i>LC_COLLATE</i> category source meets the requirements for level 3 and has been verified to produce the correct result with examples based on French, Canadian, and Danish collation order. Because it supports multi-character collating elements, it is also capable of supporting collation in codesets where a character is expressed using non-spacing characters followed by the base character (such as the ISO/IEC 6937:2001 standard).
1951	
1952	
1953	
1954	
1955	The directives that can be specified in an operand to the <b>order_start</b> keyword are based on the requirements specified in several proposed standards and in customary use. The following is a rephrasing of rules defined for “lexical ordering in English and French” by the Canadian Standards Association (the text in square brackets is rephrased):
1956	
1957	
1958	
1959	<ul style="list-style-type: none"> <li>• Once special characters [punctuation] have been removed from original strings, the ordering is determined by scanning forwards (left to right) [disregarding case and diacriticals].</li> <li>• In case of equivalence, special characters are once again removed from original strings and the ordering is determined by scanning backwards (starting from the rightmost character of the string and back), character by character [disregarding case but considering diacriticals].</li> <li>• In case of repeated equivalence, special characters are removed again from original strings and the ordering is determined by scanning forwards, character by character [considering both case and diacriticals].</li> <li>• If there is still an ordering equivalence after the first three rules have been applied, then only special characters and the position they occupy in the string are considered to determine ordering. The string that has a special character in the lowest position comes first. If two strings have a special character in the same position, the character [with the lowest collation value] comes first. In case of equality, the other special characters are considered until there is a difference or until all special characters have been exhausted.</li> </ul>
1960	
1961	
1962	
1963	
1964	
1965	
1966	
1967	
1968	
1969	It is estimated that this part of IEEE Std 1003.1-2001 covers the requirements for all European languages, and no particular problems are anticipated with Slavic or Middle East character sets.
1970	
1971	The Far East (particularly Japanese/Chinese) collations are often based on contextual information and pronunciation rules (the same ideogram can have different meanings and different pronunciations). Such collation, in general, falls outside the desired goal of IEEE Std 1003.1-2001. There are, however, several other collation rules (stroke/radical or “most common pronunciation”) that can be supported with the mechanism described here.
1972	
1973	
1974	
1975	The character order is defined by the order in which characters and elements are specified between the <b>order_start</b> and <b>order_end</b> keywords. Weights assigned to the characters and elements define the collation sequence; in the absence of weights, the character order is also the collation sequence.
1976	
1977	
1978	
1979	The <b>position</b> keyword provides the capability to consider, in a compare, the relative position of characters not subject to <b>IGNORE</b> . As an example, consider the two strings "o-ring" and "or-ing". Assuming the hyphen is subject to <b>IGNORE</b> on the first pass, the two strings compare equal, and the position of the hyphen is immaterial. On second pass, all characters except the hyphen are subject to <b>IGNORE</b> , and in the normal case the two strings would again compare equal. By taking position into account, the first collates before the second.
1980	
1981	
1982	
1983	
1984	
1985	
1986	
1987	
1988	
1989	

1990 A.7.3.3 *LC\_MONETARY*

1991 The currency symbol does not appear in *LC\_MONETARY* because it is not defined in the C locale  
 1992 of the ISO C standard.

1993 The ISO C standard limits the size of decimal points and thousands delimiters to single-byte  
 1994 values. In locales based on multi-byte coded character sets, this cannot be enforced;  
 1995 IEEE Std 1003.1-2001 does not prohibit such characters, but makes the behavior unspecified (in  
 1996 the text “In contexts where other standards ...”).

1997 The grouping specification is based on, but not identical to, the ISO C standard. The –1 indicates  
 1998 that no further grouping is performed; the equivalent of {CHAR\_MAX} in the ISO C standard.

1999 The text “the value is not available in the locale” is taken from the ISO C standard and is used  
 2000 instead of the “unspecified” text in early proposals. There is no implication that omitting these  
 2001 keywords or assigning them values of " " or –1 produces unspecified results; such omissions or  
 2002 assignments eliminate the effects described for the keyword or produce zero-length strings, as  
 2003 appropriate.

2004 The locale definition is an extension of the ISO C standard *localeconv()* specification. In  
 2005 particular, rules on how **currency\_symbol** is treated are extended to also cover **int\_curr\_symbol**,  
 2006 and **p\_sep\_by\_space** and **n\_sep\_by\_space** have been augmented with the value 2, which places  
 2007 a <space> between the sign and the symbol. This has been updated to match the 2  
 2008 ISO/IEC 9899:1999 standard requirements and is an incompatible change from UNIX 98 and the 2  
 2009 ISO POSIX-2 standard and the ISO POSIX-1:1996 standard requirements. The following table 2  
 2010 shows the result of various combinations: 2

		<b>p_sep_by_space</b>		
		<b>2</b>	<b>1</b>	<b>0</b>
<b>p_cs_precedes = 1</b>	<b>p_sign_posn = 0</b>	(\$1.25)	(\$ 1.25)	(\$1.25)
	<b>p_sign_posn = 1</b>	+ \$1.25	+\$ 1.25	+\$1.25
	<b>p_sign_posn = 2</b>	\$1.25 +	\$ 1.25+	\$1.25+
	<b>p_sign_posn = 3</b>	+ \$1.25	+\$ 1.25	+\$1.25
	<b>p_sign_posn = 4</b>	\$ +1.25	\$+ 1.25	+\$1.25
<b>p_cs_precedes = 0</b>	<b>p_sign_posn = 0</b>	(1.25 \$)	(1.25 \$)	(1.25\$)
	<b>p_sign_posn = 1</b>	+1.25 \$	+1.25 \$	+1.25\$
	<b>p_sign_posn = 2</b>	1.25\$ +	1.25 \$+	1.25\$+
	<b>p_sign_posn = 3</b>	1.25+ \$	1.25 +\$	1.25+\$
	<b>p_sign_posn = 4</b>	1.25\$ +	1.25 \$+	1.25\$+

2023 The following is an example of the interpretation of the **mon\_grouping** keyword. Assuming that  
 2024 the value to be formatted is 123 456 789 and the **mon\_thousands\_sep** is ' ', then the following  
 2025 table shows the result. The third column shows the equivalent string in the ISO C standard that  
 2026 would be used by the *localeconv()* function to accommodate this grouping.

<b>mon_grouping</b>	<b>Formatted Value</b>	<b>ISO C String</b>
3;-1	123456'789	"\3\177"
3	123'456'789	"\3"
3;2;-1	1234'56'789	"\3\2\177"
3;2	12'34'56'789	"\3\2"
-1	123456789	"\177"

2033 In these examples, the octal value of {CHAR\_MAX} is 177.

2034 IEEE Std 1003.1-2001/Cor 1-2002, item XBD/TC1/D6/6 adds a correction that permits the Euro 1  
 2035 currency symbol and addresses extensibility. The correction is stated using the term “should” 1

2036	intentionally, in order to make this a recommendation rather than a restriction on	1
2037	implementations. This allows for flexibility in implementations on how they handle future	1
2038	currency symbol additions.	1
2039	IEEE Std 1003.1-2001/Cor 1-2002, tem XBD/TC1/D6/5 is applied, adding the <b>int_[np]</b> * values	1
2040	to the POSIX locale definition of <i>LC_MONETARY</i> .	1
2041	IEEE Std 1003.1-2001/Cor 2-2004, item XBD/TC2/D6/16 is applied, updating the descriptions of	2
2042	<b>p_sep_by_space</b> , <b>n_sep_by_space</b> , <b>int_p_sep_by_space</b> , and <b>int_n_sep_by_space</b> to match the	2
2043	description of these keywords in the ISO C standard and the System Interfaces volume of	2
2044	IEEE Std 1003.1-2001, <i>localeconv()</i> .	2
2045	<b>A.7.3.4 LC_NUMERIC</b>	
2046	See the rationale for <i>LC_MONETARY</i> for a description of the behavior of grouping.	
2047	<b>A.7.3.5 LC_TIME</b>	
2048	Although certain of the conversion specifications in the POSIX locale (such as the name of the	
2049	month) are shown with initial capital letters, this need not be the case in other locales. Programs	
2050	using these conversion specifications may need to adjust the capitalization if the output is going	
2051	to be used at the beginning of a sentence.	
2052	The <i>LC_TIME</i> descriptions of <b>abday</b> , <b>day</b> , <b>mon</b> , and <b>abmon</b> imply a Gregorian style calendar (7-	
2053	day weeks, 12-month years, leap years, and so on). Formatting time strings for other types of	
2054	calendars is outside the scope of IEEE Std 1003.1-2001.	
2055	While the ISO 8601:2000 standard numbers the weekdays starting with Monday, historical	
2056	practice is to use the Sunday as the first day. Rather than change the order and introduce	
2057	potential confusion, the days must be specified beginning with Sunday; previous references to	
2058	“first day” have been removed. Note also that the Shell and Utilities volume of	
2059	IEEE Std 1003.1-2001 <i>date</i> utility supports numbering compliant with the ISO 8601:2000	
2060	standard.	
2061	As specified under <i>date</i> in the Shell and Utilities volume of IEEE Std 1003.1-2001 and <i>strftime()</i> in	
2062	the System Interfaces volume of IEEE Std 1003.1-2001, the conversion specifications	
2063	corresponding to the optional keywords consist of a modifier followed by a traditional	
2064	conversion specification (for instance, %Ex). If the optional keywords are not supported by the	
2065	implementation or are unspecified for the current locale, these modified conversion	
2066	specifications are treated as the traditional conversion specifications. For example, assume the	
2067	following keywords:	
2068	alt_digits "0th";"1st";"2nd";"3rd";"4th";"5th";\	
2069	"6th";"7th";"8th";"9th";"10th"	
2070	d_fmt "The %Od day of %B in %Y"	
2071	On July 4th 1776, the %x conversion specifications would result in "The 4th day of July	
2072	in 1776", while on July 14th 1789 it would result in "The 14 day of July in 1789". It	
2073	can be noted that the above example is for illustrative purposes only; the %O modifier is	
2074	primarily intended to provide for Kanji or Hindi digits in <i>date</i> formats.	
2075	The following is an example for Japan that supports the current plus last three Emperors and	
2076	reverts to Western style numbering for years prior to the Meiji era. The example also allows for	
2077	the custom of using a special name for the first year of an era instead of using 1. (The examples	
2078	substitute romaji where kanji should be used.)	



2079 era\_d\_fmt "%EY%mgatsu%dnichi (%a)"

2080 era "+:2:1990/01/01:+\*:Heisei:%EC%Eynen";\  
 2081 "+:1:1989/01/08:1989/12/31:Heisei:%ECgannen";\  
 2082 "+:2:1927/01/01:1989/01/07:Shouwa:%EC%Eynen";\  
 2083 "+:1:1926/12/25:1926/12/31:Shouwa:%ECgannen";\  
 2084 "+:2:1913/01/01:1926/12/24:Taishou:%EC%Eynen";\  
 2085 "+:1:1912/07/30:1912/12/31:Taishou:%ECgannen";\  
 2086 "+:2:1869/01/01:1912/07/29:Meiji:%EC%Eynen";\  
 2087 "+:1:1868/09/08:1868/12/31:Meiji:%ECgannen";\  
 2088 "-:1868:1868/09/07:-\*:%Ey"

2089 Assuming that the current date is September 21, 1991, a request to *date* or *strftime()* would yield  
 2090 the following results:

2091 %Ec - Heisei3nen9gatsu21nichi (Sat) 14:39:26  
 2092 %EC - Heisei  
 2093 %Ex - Heisei3nen9gatsu21nichi (Sat)  
 2094 %Ey - 3  
 2095 %EY - Heisei3nen

2096 Example era definitions for the Republic of China:

2097 era "+:2:1913/01/01:+\*:ChungHwaMingGuo:%EC%EyNen";\  
 2098 "+:1:1912/1/1:1912/12/31:ChungHwaMingGuo:%ECYuenNen";\  
 2099 "+:1:1911/12/31:-\*:MingChien:%EC%EyNen"

2100 Example definitions for the Christian Era:

2101 era "+:1:0001/01/01:+\*:AD:%EC %Ey";\  
 2102 "+:1:-0001/12/31:-\*:BC:%Ey %EC"

#### 2103 A.7.3.6 LC\_MESSAGES

2104 The **yesstr** and **nostr** locale keywords and the YESSTR and NOSTR *langinfo* items were formerly  
 2105 used to match user affirmative and negative responses. In IEEE Std 1003.1-2001, the **yesexpr**,  
 2106 **noexpr**, YESEXPR, and NOEXPR extended regular expressions have replaced them.  
 2107 Applications should use the general locale-based messaging facilities to issue prompting  
 2108 messages which include sample desired responses.

### 2109 A.7.4 Locale Definition Grammar

2110 There is no additional rationale provided for this section.

#### 2111 A.7.4.1 Locale Lexical Conventions

2112 There is no additional rationale provided for this section.

#### 2113 A.7.4.2 Locale Grammar

2114 There is no additional rationale provided for this section.

### 2115 A.7.5 Locale Definition Example

2116 The following is an example of a locale definition file that could be used as input to the *localedef*  
 2117 utility. It assumes that the utility is executed with the *-f* option, naming a charmap file with (at  
 2118 least) the following content:

```
2119     CHARMAP
2120     <space>      \x20
2121     <dollar>      \x24
2122     <A>           \101
2123     <a>           \141
2124     <A-acute>     \346
2125     <a-acute>     \365
2126     <A-grave>     \300
2127     <a-grave>     \366
2128     <b>           \142
2129     <C>           \103
2130     <c>           \143
2131     <c-cedilla>   \347
2132     <d>           \x64
2133     <H>           \110
2134     <h>           \150
2135     <eszet>       \xb7
2136     <s>           \x73
2137     <z>           \x7a
2138     END CHARMAP
```

2139 It should not be taken as complete or to represent any actual locale, but only to illustrate the  
 2140 syntax.

```
2141     #
2142     LC_CTYPE
2143     lower  <a>;<b>;<c>;<c-cedilla>;<d>;...;<z>
2144     upper  A;B;C;Ç;...;Z
2145     space  \x20;\x09;\x0a;\x0b;\x0c;\x0d
2146     blank  \040;\011
2147     toupper (<a>,<A>);(b,B);(c,C);(ç,Ç);(d,D);(z,Z)
2148     END LC_CTYPE
2149     #
2150     LC_COLLATE
2151     #
2152     # The following example of collation is based on
2153     # Canadian standard Z243.4.1-1998, "Canadian Alphanumeric
2154     # Ordering Standard for Character Sets of CSA Z234.4 Standard".
2155     # (Other parts of this example locale definition file do not
2156     # purport to relate to Canada, or to any other real culture.)
2157     # The proposed standard defines a 4-weight collation, such that
2158     # in the first pass, characters are compared without regard to
2159     # case or accents; in the second pass, backwards-compare without
2160     # regard to case; in the third pass, forwards-compare without
2161     # regard to diacriticals. In the 3 first passes, non-alphabetic
2162     # characters are ignored; in the fourth pass, only special
2163     # characters are considered, such that "The string that has a
2164     # special character in the lowest position comes first. If two
```

```

2165      # strings have a special character in the same position, the
2166      # collation value of the special character determines ordering.
2167      #
2168      # Only a subset of the character set is used here; mostly to
2169      # illustrate the set-up.
2170      #
2171      collating-symbol <NULL>
2172      collating-symbol <LOW_VALUE>
2173      collating-symbol <LOWER-CASE>
2174      collating-symbol <SUBSCRIPT-LOWER>
2175      collating-symbol <SUPERSCRIPT-LOWER>
2176      collating-symbol <UPPER-CASE>
2177      collating-symbol <NO-ACCENT>
2178      collating-symbol <PECULIAR>
2179      collating-symbol <LIGATURE>
2180      collating-symbol <ACUTE>
2181      collating-symbol <GRAVE>
2182      # Further collating-symbols follow.
2183      #
2184      # Properly, the standard does not include any multi-character
2185      # collating elements; the one below is added for completeness.
2186      #
2187      collating_element <ch> from "<c><h>"
2188      collating_element <CH> from "<C><H>"
2189      collating_element <Ch> from "<C><h>"
2190      #
2191      order_start forward;backward;forward;forward,position
2192      #
2193      # Collating symbols are specified first in the sequence to allocate
2194      # basic collation values to them, lower than that of any character.
2195      <NULL>
2196      <LOW_VALUE>
2197      <LOWER-CASE>
2198      <SUBSCRIPT-LOWER>
2199      <SUPERSCRIPT-LOWER>
2200      <UPPER-CASE>
2201      <NO-ACCENT>
2202      <PECULIAR>
2203      <LIGATURE>
2204      <ACUTE>
2205      <GRAVE>
2206      <RING-ABOVE>
2207      <DIAERESIS>
2208      <TILDE>
2209      # Further collating symbols are given a basic collating value here.
2210      #
2211      # Here follow special characters.
2212      <space>          IGNORE;IGNORE;IGNORE;<space>
2213      # Other special characters follow here.
2214      #
2215      # Here follow the regular characters.
2216      <a>              <a>;<NO-ACCENT>;<LOWER-CASE>;IGNORE

```

```

2217      <A>          <a>; <NO-ACCENT>; <UPPER-CASE>; IGNORE
2218      <a-acute>    <a>; <ACUTE>; <LOWER-CASE>; IGNORE
2219      <A-acute>    <a>; <ACUTE>; <UPPER-CASE>; IGNORE
2220      <a-grave>    <a>; <GRAVE>; <LOWER-CASE>; IGNORE
2221      <A-grave>    <a>; <GRAVE>; <UPPER-CASE>; IGNORE
2222      <ae>          "<a><e>" ; "<LIGATURE><LIGATURE>" ; \
2223                  "<LOWER-CASE><LOWER-CASE>" ; IGNORE
2224      <AE>          "<a><e>" ; "<LIGATURE><LIGATURE>" ; \
2225                  "<UPPER-CASE><UPPER-CASE>" ; IGNORE
2226      <b>          <b>; <NO-ACCENT>; <LOWER-CASE>; IGNORE
2227      <B>          <b>; <NO-ACCENT>; <UPPER-CASE>; IGNORE
2228      <c>          <c>; <NO-ACCENT>; <LOWER-CASE>; IGNORE
2229      <C>          <c>; <NO-ACCENT>; <UPPER-CASE>; IGNORE
2230      <ch>         <ch>; <NO-ACCENT>; <LOWER-CASE>; IGNORE
2231      <Ch>         <ch>; <NO-ACCENT>; <PECULIAR>; IGNORE
2232      <CH>         <ch>; <NO-ACCENT>; <UPPER-CASE>; IGNORE
2233      #
2234      # As an example, the strings "Bach" and "bach" could be encoded (for
2235      # compare purposes) as:
2236      # "Bach"      <b>; <a>; <ch>; <LOW_VALUE>; <NO-ACCENT>; \
2237      #              <NO-ACCENT>; <LOW_VALUE>; <UPPER-CASE>; <LOWER-CASE>; \
2238      #              <LOWER-CASE>; <NULL>
2239      # "bach"      <b>; <a>; <ch>; <LOW_VALUE>; <NO-ACCENT>; \
2240      #              <NO-ACCENT>; <LOW_VALUE>; <LOWER-CASE>; <LOWER-CASE>; \
2241      #              <LOWER-CASE>; <NULL>
2242      #
2243      # The two strings are equal in pass 1 and 2, but differ in pass 3.
2244      #
2245      # Further characters follow.
2246      #
2247      UNDEFINED      IGNORE; IGNORE; IGNORE; IGNORE
2248      #
2249      order_end
2250      #
2251      END LC_COLLATE
2252      #
2253      LC_MONETARY
2254      int_curr_symbol      "USD "
2255      currency_symbol      "$"
2256      mon_decimal_point    "."
2257      mon_grouping         3;0
2258      positive_sign        ""
2259      negative_sign        "- "
2260      p_cs_precedes         1
2261      n_sign_posn           0
2262      END LC_MONETARY
2263      #
2264      LC_NUMERIC
2265      copy "US_en.ASCII"
2266      END LC_NUMERIC
2267      #
2268      LC_TIME

```

```

2269     abday    "Sun"; "Mon"; "Tue"; "Wed"; "Thu"; "Fri"; "Sat "
2270     #
2271     day      "Sunday"; "Monday"; "Tuesday"; "Wednesday"; \
2272             "Thursday"; "Friday"; "Saturday"
2273     #
2274     abmon    "Jan"; "Feb"; "Mar"; "Apr"; "May"; "Jun"; \
2275             "Jul"; "Aug"; "Sep"; "Oct"; "Nov"; "Dec "
2276     #
2277     mon      "January"; "February"; "March"; "April"; \
2278             "May"; "June"; "July"; "August"; "September"; \
2279             "October"; "November"; "December"
2280     #
2281     d_t_fmt  "%a %b %d %T %Z %Y\n"
2282     END LC_TIME
2283     #
2284     LC_MESSAGES
2285     yesexpr  "^[yY][[:alpha:]]*)" | (OK) "
2286     #
2287     noexpr   "^[nN][[:alpha:]]*"
2288     END LC_MESSAGES

```

## 2289 A.8 Environment Variables

### 2290 A.8.1 Environment Variable Definition

2291 The variable *environ* is not intended to be declared in any header, but rather to be declared by the  
 2292 user for accessing the array of strings that is the environment. This is the traditional usage of the  
 2293 symbol. Putting it into a header could break some programs that use the symbol for their own  
 2294 purposes.

2295 The decision to restrict conforming systems to the use of digits, uppercase letters, and  
 2296 underscores for environment variable names allows applications to use lowercase letters in their  
 2297 environment variable names without conflicting with any conforming system.

2298 In addition to the obvious conflict with the shell syntax for positional parameter substitution,  
 2299 some historical applications (including some shells) exclude names with leading digits from the  
 2300 environment.

### 2301 A.8.2 Internationalization Variables

2302 Utilities conforming to the Shell and Utilities volume of IEEE Std 1003.1-2001 and written in 2  
 2303 standard C can access the locale variables by issuing the following call: 2

```
2304     setlocale(LC_ALL, "")
```

2305 If this were omitted, the ISO C standard specifies that the C locale would be used.

2306 The DESCRIPTION of *setlocale()* requires that when setting all categories of a locale, if the value 2  
 2307 of any of the environment variable searches yields a locale that is not supported (and non-null), 2  
 2308 the *setlocale()* function returns a null pointer and the locale of the process is unchanged. 2

2309 For the standard utilities, if any of the environment variables are invalid, it makes sense to 2  
 2310 default to an implementation-defined, consistent locale environment. It is more confusing for a 2  
 2311 user to have partial settings occur in case of a mistake. All utilities would then behave in one  
 2312 language/cultural environment. Furthermore, it provides a way of forcing the whole

environment to be the implementation-defined default. Disastrous results could occur if a pipeline of utilities partially uses the environment variables in different ways. In this case, it would be appropriate for utilities that use *LANG* and related variables to exit with an error if any of the variables are invalid. For example, users typing individual commands at a terminal might want *date* to work if *LC\_MONETARY* is invalid as long as *LC\_TIME* is valid. Since these are conflicting reasonable alternatives, IEEE Std 1003.1-2001 leaves the results unspecified if the locale environment variables would not produce a complete locale matching the specification of the user.

The locale settings of individual categories cannot be truly independent and still guarantee correct results. For example, when collating two strings, characters must first be extracted from each string (governed by *LC\_CTYPE*) before being mapped to collating elements (governed by *LC\_COLLATE*) for comparison. That is, if *LC\_CTYPE* is causing parsing according to the rules of a large, multi-byte code set (potentially returning 20 000 or more distinct character codeset values), but *LC\_COLLATE* is set to handle only an 8-bit codeset with 256 distinct characters, meaningful results are obviously impossible.

The *LC\_MESSAGES* variable affects the language of messages generated by the standard utilities.

The description of the environment variable names starting with the characters “LC\_” acknowledges the fact that the interfaces presented may be extended as new international functionality is required. In the ISO C standard, names preceded by “LC\_” are reserved in the name space for future categories.

To avoid name clashes, new categories and environment variables are divided into two classifications: “implementation-independent” and “implementation-defined”.

Implementation-independent names will have the following format:

*LC\_NAME*

where *NAME* is the name of the new category and environment variable. Capital letters must be used for implementation-independent names.

Implementation-defined names must be in lowercase letters, as below:

*LC\_name*

### A.8.3 Other Environment Variables

#### COLUMNS, LINES

The default values for the number of column positions, *COLUMNS*, and screen height, *LINES*, are unspecified because historical implementations use different methods to determine values corresponding to the size of the screen in which the utility is run. This size is typically known to the implementation through the value of *TERM*, or by more elaborate methods such as extensions to the *stty* utility or knowledge of how the user is dynamically resizing windows on a bit-mapped display terminal. Users should not need to set these variables in the environment unless there is a specific reason to override the default behavior of the implementation, such as to display data in an area arbitrarily smaller than the terminal or window. Values for these variables that are not decimal integers greater than zero are implicitly undefined values; it is unnecessary to enumerate all of the possible values outside of the acceptable set.

**LOGNAME**

In most implementations, the value of such a variable is easily forged, so security-critical applications should rely on other means of determining user identity. *LOGNAME* is required to be constructed from the portable filename character set for reasons of interchange. No diagnostic condition is specified for violating this rule, and no requirement for enforcement exists. The intent of the requirement is that if extended characters are used, the “guarantee” of portability implied by a standard is void.

**PATH**

Many historical implementations of the Bourne shell do not interpret a trailing colon to represent the current working directory and are thus non-conforming. The C Shell and the KornShell conform to IEEE Std 1003.1-2001 on this point. The usual name of dot may also be used to refer to the current working directory.

Many implementations historically have used a default value of */bin* and */usr/bin* for the *PATH* variable. IEEE Std 1003.1-2001 does not mandate this default path be identical to that retrieved from *getconf \_CS\_PATH* because it is likely that the standardized utilities may be provided in another directory separate from the directories used by some historical applications.

**SHELL**

The *SHELL* variable names the preferred shell of the user; it is a guide to applications. There is no direct requirement that that shell conform to IEEE Std 1003.1-2001; that decision should rest with the user. It is the intention of the standard developers that alternative shells be permitted, if the user chooses to develop or acquire one. An operating system that builds its shell into the “kernel” in such a manner that alternative shells would be impossible does not conform to the spirit of IEEE Std 1003.1-2001.

**TZ**

The quoted form of the timezone variable allows timezone names of the form UTC+1 (or any name that contains the character plus (+), the character minus (-), or digits), which may be appropriate for countries that do not have an official timezone name. It would be coded as <UTC+1>+1<UTC+2>, which would cause *std* to have a value of UTC+1 and *dst* a value of UTC+2, each with a length of 5 characters. This does not appear to conflict with any existing usage. The characters ‘<’ and ‘>’ were chosen for quoting because they are easier to parse visually than a quoting character that does not provide some sense of bracketing (and in a string like this, such bracketing is helpful). They were also chosen because they do not need special treatment when assigning to the *TZ* variable. Users are often confused by embedding quotes in a string. Because ‘<’ and ‘>’ are meaningful to the shell, the whole string would have to be quoted, but that is easily explained. (Parentheses would have presented the same problems.) Although the ‘>’ symbol could have been permitted in the string by either escaping it or doubling it, it seemed of little value to require that. This could be provided as an extension if there was a need. Timezone names of this new form lead to a requirement that the value of `{_POSIX_TZNAME_MAX}` change from 3 to 6.

Since the *TZ* environment variable is usually inherited by all applications started by a user after the value of the *TZ* environment variable is changed and since many applications run using the C or POSIX locale, using characters that are not in the portable character set in the *std* and *dst* fields could cause unexpected results.

The format of the *TZ* environment variable is changed in Issue 6 to allow for the quoted form, as defined in previous versions of the ISO POSIX-1 standard.

2399 IEEE Std 1003.1-2001/Cor 1-2002, item XBD/TC1/D6/7 is applied, adding the *ctime\_r()* and 1  
 2400 *localtime\_r()* functions to the list of functions that use the TZ environment variable. 1

## 2401 A.9 Regular Expressions

2402 Rather than repeating the description of REs for each utility supporting REs, the standard  
 2403 developers preferred a common, comprehensive description of regular expressions in one place.  
 2404 The most common behavior is described here, and exceptions or extensions to this are  
 2405 documented for the respective utilities, as appropriate.

2406 The BRE corresponds to the *ed* or historical *grep* type, and the ERE corresponds to the historical  
 2407 *egrep* type (now *grep -E*).

2408 The text is based on the *ed* description and substantially modified, primarily to aid developers  
 2409 and others in the understanding of the capabilities and limitations of REs. Much of this was  
 2410 influenced by internationalization requirements.

2411 It should be noted that the definitions in this section do not cover the *tr* utility; the *tr* syntax does  
 2412 not employ REs.

2413 The specification of REs is particularly important to internationalization because pattern  
 2414 matching operations are very basic operations in business and other operations. The syntax and  
 2415 rules of REs are intended to be as intuitive as possible to make them easy to understand and use.  
 2416 The historical rules and behavior do not provide that capability to non-English language users,  
 2417 and do not provide the necessary support for commonly used characters and language  
 2418 constructs. It was necessary to provide extensions to the historical RE syntax and rules to  
 2419 accommodate other languages.

2420 As they are limited to bracket expressions, the rationale for these modifications is in the Base  
 2421 Definitions volume of IEEE Std 1003.1-2001, Section 9.3.5, RE Bracket Expression.

### 2422 A.9.1 Regular Expression Definitions

2423 It is possible to determine what strings correspond to subexpressions by recursively applying  
 2424 the leftmost longest rule to each subexpression, but only with the proviso that the overall match  
 2425 is leftmost longest. For example, matching "*\(ac\*\)c\*d[ac]\*\1*" against *acdacaaa* matches  
 2426 *acdacaaa* (with *\1=a*); simply matching the longest match for "*\(ac\*\)*" would yield *\1=ac*, but  
 2427 the overall match would be smaller (*acdac*). Conceptually, the implementation must examine  
 2428 every possible match and among those that yield the leftmost longest total matches, pick the one  
 2429 that does the longest match for the leftmost subexpression, and so on. Note that this means that  
 2430 matching by subexpressions is context-dependent: a subexpression within a larger RE may  
 2431 match a different string from the one it would match as an independent RE, and two instances of  
 2432 the same subexpression within the same larger RE may match different lengths even in similar  
 2433 sequences of characters. For example, in the ERE "*(a.\*b)(a.\*b)*", the two identical  
 2434 subexpressions would match four and six characters, respectively, of *accbaccccb*.

2435 The definition of single character has been expanded to include also collating elements  
 2436 consisting of two or more characters; this expansion is applicable only when a bracket  
 2437 expression is included in the BRE or ERE. An example of such a collating element may be the  
 2438 Dutch *ij*, which collates as a 'y'. In some encodings, a ligature "i with j" exists as a character  
 2439 and would represent a single-character collating element. In another encoding, no such ligature  
 2440 exists, and the two-character sequence *ij* is defined as a multi-character collating element.  
 2441 Outside brackets, the *ij* is treated as a two-character RE and matches the same characters in a  
 2442 string. Historically, a bracket expression only matched a single character. The ISO POSIX-2: 1993  
 2443 standard required bracket expressions like "*^[[:lower:]]*" to match multi-character collating



elements such as "ij". However, this requirement led to behavior that many users did not expect and that could not feasibly be mimicked in user code, and it was rarely if ever implemented correctly. The current standard leaves it unspecified whether a bracket expression matches a multi-character collating element, allowing both historical and ISO POSIX-2:1993 standard implementations to conform.

Also, in the current standard, it is unspecified whether character class expressions like "[[:lower:]]" can include multi-character collating elements like "ij"; hence "[[:lower:]]" can match "ij", and "[^[:lower:]]" can fail to match "ij". Common practice is for a character class expression to match a collating element if it matches the collating element's first character.

## A.9.2 Regular Expression General Requirements

The definition of which sequence is matched when several are possible is based on the leftmost-longest rule historically used by deterministic recognizers. This rule is easier to define and describe, and arguably more useful, than the first-match rule historically used by non-deterministic recognizers. It is thought that dependencies on the choice of rule are rare; carefully contrived examples are needed to demonstrate the difference.

A formal expression of the leftmost-longest rule is:

The search is performed as if all possible suffixes of the string were tested for a prefix matching the pattern; the longest suffix containing a matching prefix is chosen, and the longest possible matching prefix of the chosen suffix is identified as the matching sequence.

Historically, most RE implementations only match lines, not strings. However, that is more an effect of the usage than of an inherent feature of REs themselves. Consequently, IEEE Std 1003.1-2001 does not regard <newline>s as special; they are ordinary characters, and both a period and a non-matching list can match them. Those utilities (like *grep*) that do not allow <newline>s to match are responsible for eliminating any <newline> from strings before matching against the RE. The *regcomp()* function, however, can provide support for such processing without violating the rules of this section.

Some implementations of *egrep* have had very limited flexibility in handling complex EREs. IEEE Std 1003.1-2001 does not attempt to define the complexity of a BRE or ERE, but does place a lower limit on it—any RE must be handled, as long as it can be expressed in 256 bytes or less. (Of course, this does not place an upper limit on the implementation.) There are historical programs using a non-deterministic-recognizer implementation that should have no difficulty with this limit. It is possible that a good approach would be to attempt to use the faster, but more limited, deterministic recognizer for simple expressions and to fall back on the non-deterministic recognizer for those expressions requiring it. Non-deterministic implementations must be careful to observe the rules on which match is chosen; the longest match, not the first match, starting at a given character is used.

The term “invalid” highlights a difference between this section and some others: IEEE Std 1003.1-2001 frequently avoids mandating of errors for syntax violations because they can be used by implementors to trigger extensions. However, the authors of the internationalization features of REs wanted to mandate errors for certain conditions to identify usage problems or non-portable constructs. These are identified within this rationale as appropriate. The remaining syntax violations have been left implicitly or explicitly undefined. For example, the BRE construct "\{1,2,3\}" does not comply with the grammar. A conforming application cannot rely on it producing an error nor matching the literal characters "\{1,2,3\}".

2490 The term “undefined” was used in favor of “unspecified” because many of the situations are  
 2491 considered errors on some implementations, and the standard developers considered that  
 2492 consistency throughout the section was preferable to mixing undefined and unspecified.

### 2493 **A.9.3 Basic Regular Expressions**

2494 There is no additional rationale provided for this section.

#### 2495 *A.9.3.1 BREs Matching a Single Character or Collating Element*

2496 There is no additional rationale provided for this section.

#### 2497 *A.9.3.2 BRE Ordinary Characters*

2498 There is no additional rationale provided for this section.

#### 2499 *A.9.3.3 BRE Special Characters*

2500 There is no additional rationale provided for this section.

#### 2501 *A.9.3.4 Periods in BREs*

2502 There is no additional rationale provided for this section.

#### 2503 *A.9.3.5 RE Bracket Expression*

2504 Range expressions are, historically, an integral part of REs. However, the requirements of  
 2505 “natural language behavior” and portability do conflict. In the POSIX locale, ranges must be  
 2506 treated according to the collating sequence and include such characters that fall within the range  
 2507 based on that collating sequence, regardless of character values. In other locales, ranges have  
 2508 unspecified behavior.

2509 Some historical implementations allow range expressions where the ending range point of one  
 2510 range is also the starting point of the next (for instance, "[a-m-o]"). This behavior should not  
 2511 be permitted, but to avoid breaking historical implementations, it is now *undefined* whether it is a  
 2512 valid expression and how it should be interpreted.

2513 Current practice in *awk* and *lex* is to accept escape sequences in bracket expressions as per the  
 2514 Base Definitions volume of IEEE Std 1003.1-2001, Table 5-1, Escape Sequences and Associated  
 2515 Actions, while the normal ERE behavior is to regard such a sequence as consisting of two  
 2516 characters. Allowing the *awk/lex* behavior in EREs would change the normal behavior in an  
 2517 unacceptable way; it is expected that *awk* and *lex* will decode escape sequences in EREs before  
 2518 passing them to *regcomp()* or comparable routines. Each utility describes the escape sequences it  
 2519 accepts as an exception to the rules in this section; the list is not the same, for historical reasons.

2520 As noted previously, the new syntax and rules have been added to accommodate other  
 2521 languages than English. The remainder of this section describes the rationale for these  
 2522 modifications.

2523 In the POSIX locale, a regular expression that starts with a range expression matches a set of  
 2524 strings that are contiguously sorted, but this is not necessarily true in other locales. For example,  
 2525 a French locale might have the following behavior:

```
2526 $ ls
2527 alpha  Alpha  estimé  ESTIMÉ  été  eurêka
2528 $ ls [a-e]*
2529 alpha  Alpha  estimé  eurêka
```

Such disagreements between matching and contiguous sorting are unavoidable because POSIX sorting cannot be implemented in terms of a deterministic finite-state automaton (DFA), but range expressions by design are implementable in terms of DFAs.

Historical implementations used native character order to interpret range expressions. The ISO POSIX-2:1993 standard instead required collating element order (CEO): the order that collating elements were specified between the **order\_start** and **order\_end** keywords in the *LC\_COLLATE* category of the current locale. CEO had some advantages in portability over the native character order, but it also had some disadvantages:

- CEO could not feasibly be mimicked in user code, leading to inconsistencies between POSIX matchers and matchers in popular user programs like Emacs, *ksh*, and Perl.
- CEO caused range expressions to match accented and capitalized letters contrary to many users' expectations. For example, "[a-e]" typically matched both 'E' and 'á' but neither 'A' nor 'é'.
- CEO was not consistent across implementations. In practice, CEO was often less portable than native character order. For example, it was common for the CEOs of two implementation-supplied locales to disagree, even if both locales were named "da\_DK".

Because of these problems, some implementations of regular expressions continued to use native character order. Others used the collation sequence, which is more consistent with sorting than either CEO or native order, but which departs further from the traditional POSIX semantics because it generally requires "[a-e]" to match either 'A' or 'E' but not both. As a result of this kind of implementation variation, programmers who wanted to write portable regular expressions could not rely on the ISO POSIX-2:1993 standard guarantees in practice.

While revising the standard, lengthy consideration was given to proposals to attack this problem by adding an API for querying the CEO to allow user-mode matchers, but none of these proposals had implementation experience and none achieved consensus. Leaving the standard alone was also considered, but rejected due to the problems described above.

The current standard leaves unspecified the behavior of a range expression outside the POSIX locale. This makes it clearer that conforming applications should avoid range expressions outside the POSIX locale, and it allows implementations and compatible user-mode matchers to interpret range expressions using native order, CEO, collation sequence, or other, more advanced techniques. The concerns which led to this change were raised in IEEE PASC interpretation 1003.2 #43 and others, and related to ambiguities in the specification of how multi-character collating elements should be handled in range expressions. These ambiguities had led to multiple interpretations of the specification, in conflicting ways, which led to varying implementations. As noted above, efforts were made to resolve the differences, but no solution has been found that would be specific enough to allow for portable software while not invalidating existing implementations.

The standard developers recognize that collating elements are important, such elements being common in several European languages; for example, 'ch' or 'll' in traditional Spanish; 'aa' in several Scandinavian languages. Existing internationalized implementations have processed, and continue to process, these elements in range expressions. Efforts are expected to continue in the future to find a way to define the behavior of these elements precisely and portably.

The ISO POSIX-2:1993 standard required "[b-a]" to be an invalid expression in the POSIX locale, but this requirement has been relaxed in this version of the standard so that "[b-a]" can instead be treated as a valid expression that does not match any string.

## 2575 A.9.3.6 BREs Matching Multiple Characters

2576 The limit of nine back-references to subexpressions in the RE is based on the use of a single-digit  
 2577 identifier; increasing this to multiple digits would break historical applications. This does not  
 2578 imply that only nine subexpressions are allowed in REs. The following is a valid BRE with ten  
 2579 subexpressions:

2580 `\(\(\(ab\) *c\) *d\)\(ef\) *(gh\)\{2\}\(ij\) *(kl\) *(mn\) *(op\) *(qr\) *`

2581 The standard developers regarded the common historical behavior, which supported "`\n*`", but  
 2582 not "`\n\{min,max\}`", "`\(...\)*`", or "`\(...\)\{min,max\}`", as a non-intentional  
 2583 result of a specific implementation, and they supported both duplication and interval  
 2584 expressions following subexpressions and back-references.

2585 The changes to the processing of the back-reference expression remove an unspecified or  
 2586 ambiguous behavior in the Shell and Utilities volume of IEEE Std 1003.1-2001, aligning it with  
 2587 the requirements specified for the *regcomp()* expression, and is the result of PASC Interpretation  
 2588 1003.2-92 #43 submitted for the ISO POSIX-2: 1993 standard.

## 2589 A.9.3.7 BRE Precedence

2590 There is no additional rationale provided for this section.

## 2591 A.9.3.8 BRE Expression Anchoring

2592 Often, the dollar sign is viewed as matching the ending <newline> in text files. This is not  
 2593 strictly true; the <newline> is typically eliminated from the strings to be matched, and the dollar  
 2594 sign matches the terminating null character.

2595 The ability of '^', '\$', and '\*' to be non-special in certain circumstances may be confusing to  
 2596 some programmers, but this situation was changed only in a minor way from historical practice  
 2597 to avoid breaking many historical scripts. Some consideration was given to making the use of  
 2598 the anchoring characters undefined if not escaped and not at the beginning or end of strings.  
 2599 This would cause a number of historical BREs, such as "`2^10`", "`$HOME`", and "`$1.35`", that  
 2600 relied on the characters being treated literally, to become invalid.

2601 However, one relatively uncommon case was changed to allow an extension used on some  
 2602 implementations. Historically, the BREs "`^foo`" and "`\(^foo\)`" did not match the same  
 2603 string, despite the general rule that subexpressions and entire BREs match the same strings. To  
 2604 increase consensus, IEEE Std 1003.1-2001 has allowed an extension on some implementations to  
 2605 treat these two cases in the same way by declaring that anchoring *may* occur at the beginning or  
 2606 end of a subexpression. Therefore, portable BREs that require a literal circumflex at the  
 2607 beginning or a dollar sign at the end of a subexpression must escape them. Note that a BRE such  
 2608 as "`a\(^bc\)`" will either match "`a^bc`" or nothing on different systems under the rules.

2609 ERE anchoring has been different from BRE anchoring in all historical systems. An unescaped  
 2610 anchor character has never matched its literal counterpart outside a bracket expression. Some  
 2611 implementations treated "`foo$bar`" as a valid expression that never matched anything; others  
 2612 treated it as invalid. IEEE Std 1003.1-2001 mandates the former, valid unmatched behavior.

2613 Some implementations have extended the BRE syntax to add alternation. For example, the  
 2614 subexpression "`\(foo$|bar\)`" would match either "`foo`" at the end of the string or "`bar`"  
 2615 anywhere. The extension is triggered by the use of the undefined "`\|`" sequence. Because the  
 2616 BRE is undefined for portable scripts, the extending system is free to make other assumptions,  
 2617 such that the '\$' represents the end-of-line anchor in the middle of a subexpression. If it were  
 2618 not for the extension, the '\$' would match a literal dollar sign under the rules.

#### 2619 **A.9.4 Extended Regular Expressions**

2620 As with BREs, the standard developers decided to make the interpretation of escaped ordinary  
2621 characters undefined.

2622 The right parenthesis is not listed as an ERE special character because it is only special in the  
2623 context of a preceding left parenthesis. If found without a preceding left parenthesis, the right  
2624 parenthesis has no special meaning.

2625 The interval expression, " $\{m,n\}$ ", has been added to EREs. Historically, the interval expression  
2626 has only been supported in some ERE implementations. The standard developers estimated that  
2627 the addition of interval expressions to EREs would not decrease consensus and would also make  
2628 BREs more of a subset of EREs than in many historical implementations.

2629 It was suggested that, in addition to interval expressions, back-references ( $\backslash n$ ) should also be  
2630 added to EREs. This was rejected by the standard developers as likely to decrease consensus.

2631 In historical implementations, multiple duplication symbols are usually interpreted from left to  
2632 right and treated as additive. As an example, " $a^*b$ " matches zero or more instances of ' $a$ '  
2633 followed by a ' $b$ '. In IEEE Std 1003.1-2001, multiple duplication symbols are undefined; that is,  
2634 they cannot be relied upon for conforming applications. One reason for this is to provide some  
2635 scope for future enhancements.

2636 The precedence of operations differs between EREs and those in *lex*; in *lex*, for historical reasons,  
2637 interval expressions have a lower precedence than concatenation.

##### 2638 **A.9.4.1 EREs Matching a Single Character or Collating Element**

2639 There is no additional rationale provided for this section.

##### 2640 **A.9.4.2 ERE Ordinary Characters**

2641 There is no additional rationale provided for this section.

##### 2642 **A.9.4.3 ERE Special Characters**

2643 There is no additional rationale provided for this section.

##### 2644 **A.9.4.4 Periods in EREs**

2645 There is no additional rationale provided for this section.

##### 2646 **A.9.4.5 ERE Bracket Expression**

2647 There is no additional rationale provided for this section.

##### 2648 **A.9.4.6 EREs Matching Multiple Characters**

2649 There is no additional rationale provided for this section.

##### 2650 **A.9.4.7 ERE Alternation**

2651 There is no additional rationale provided for this section.

2652 A.9.4.8 *ERE Precedence*

2653 There is no additional rationale provided for this section.

2654 A.9.4.9 *ERE Expression Anchoring*

2655 There is no additional rationale provided for this section.

2656 A.9.5 **Regular Expression Grammar**

2657 The grammars are intended to represent the range of acceptable syntaxes available to  
 2658 conforming applications. There are instances in the text where undefined constructs are  
 2659 described; as explained previously, these allow implementation extensions. There is no intended  
 2660 requirement that an implementation extension must somehow fit into the grammars shown  
 2661 here.

2662 The BRE grammar does not permit L\_ANCHOR or R\_ANCHOR inside "\(" and "\)" (which  
 2663 implies that '^' and '\$' are ordinary characters). This reflects the semantic limits on the  
 2664 application, as noted in the Base Definitions volume of IEEE Std 1003.1-2001, Section 9.3.8, BRE  
 2665 Expression Anchoring. Implementations are permitted to extend the language to interpret '^'  
 2666 and '\$' as anchors in these locations, and as such, conforming applications cannot use  
 2667 unescaped '^' and '\$' in positions inside "\(" and "\)" that might be interpreted as anchors.

2668 The ERE grammar does not permit several constructs that the Base Definitions volume of  
 2669 IEEE Std 1003.1-2001, Section 9.4.2, ERE Ordinary Characters and the Base Definitions volume of  
 2670 IEEE Std 1003.1-2001, Section 9.4.3, ERE Special Characters specify as having undefined results:

- 2671 • ORD\_CHAR preceded by '\'
- 2672 • *ERE\_dupl\_symbol*(s) appearing first in an ERE, or immediately following '|', '^', or '('
- 2673 • '{' not part of a valid *ERE\_dupl\_symbol*
- 2674 • '|' appearing first or last in an ERE, or immediately following '|' or '(', or immediately  
 2675 preceding ')'

2676 Implementations are permitted to extend the language to allow these. Conforming applications  
 2677 cannot use such constructs.

2678 A.9.5.1 *BRE/ERE Grammar Lexical Conventions*

2679 There is no additional rationale provided for this section.

2680 A.9.5.2 *RE and Bracket Expression Grammar*

2681 The removal of the *Back\_open\_paren Back\_close\_paren* option from the *nondupl\_RE* specification is  
 2682 the result of PASC Interpretation 1003.2-92 #43 submitted for the ISO POSIX-2: 1993 standard.  
 2683 Although the grammar required support for null subexpressions, this section does not describe  
 2684 the meaning of, and historical practice did not support, this construct.

2685 A.9.5.3 *ERE Grammar*

2686 There is no additional rationale provided for this section.

## 2687 A.10 Directory Structure and Devices

### 2688 A.10.1 Directory Structure and Files

2689 A description of the historical **/usr/tmp** was omitted, removing any concept of differences in  
 2690 emphasis between the **/** and **/usr** directories. The descriptions of **/bin**, **/usr/bin**, **/lib**, and **/usr/lib**  
 2691 were omitted because they are not useful for applications. In an early draft, a distinction was  
 2692 made between system and application directory usage, but this was not found to be useful.

2693 The directories **/** and **/dev** are included because the notion of a hierarchical directory structure is  
 2694 key to other information presented elsewhere in IEEE Std 1003.1-2001. In early drafts, it was  
 2695 argued that special devices and temporary files could conceivably be handled without a  
 2696 directory structure on some implementations. For example, the system could treat the characters  
 2697 `" /tmp "` as a special token that would store files using some non-POSIX file system structure.  
 2698 This notion was rejected by the standard developers, who required that all the files in this  
 2699 section be implemented via POSIX file systems.

2700 The **/tmp** directory is retained in IEEE Std 1003.1-2001 to accommodate historical applications  
 2701 that assume its availability. Implementations are encouraged to provide suitable directory  
 2702 names in the environment variable *TMPDIR* and applications are encouraged to use the contents  
 2703 of *TMPDIR* for creating temporary files.

2704 The standard files **/dev/null** and **/dev/tty** are required to be both readable and writable to allow  
 2705 applications to have the intended historical access to these files.

2706 The standard file **/dev/console** has been added for alignment with the Single UNIX Specification.

### 2707 A.10.2 Output Devices and Terminal Types

2708 IEEE Std 1003.1-2001/Cor 2-2004, item XBD/TC2/D6/17 is applied, making it clear that the 2  
 2709 requirements for documenting terminal support are in the system documentation. 2

## 2710 A.11 General Terminal Interface

2711 If the implementation does not support this interface on any device types, it should behave as if  
 2712 it were being used on a device that is not a terminal device (in most cases *errno* will be set to  
 2713 [ENOTTY] on return from functions defined by this interface). This is based on the fact that  
 2714 many applications are written to run both interactively and in some non-interactive mode, and  
 2715 they adapt themselves at runtime. Requiring that they all be modified to test an environment  
 2716 variable to determine whether they should try to adapt is unnecessary. On a system that  
 2717 provides no general terminal interface, providing all the entry points as stubs that return  
 2718 [ENOTTY] (or an equivalent, as appropriate) has the same effect and requires no changes to the  
 2719 application.

2720 Although the needs of both interface implementors and application developers were addressed  
 2721 throughout IEEE Std 1003.1-2001, this section pays more attention to the needs of the latter. This  
 2722 is because, while many aspects of the programming interface can be hidden from the user by the  
 2723 application developer, the terminal interface is usually a large part of the user interface.  
 2724 Although to some extent the application developer can build missing features or work around  
 2725 inappropriate ones, the difficulties of doing that are greater in the terminal interface than  
 2726 elsewhere. For example, efficiency prohibits the average program from interpreting every  
 2727 character passing through it in order to simulate character erase, line kill, and so on. These  
 2728 functions should usually be done by the operating system, possibly at the interrupt level.

The *tc\*()* functions were introduced as a way of avoiding the problems inherent in the traditional *ioctl()* function and in variants of it that were proposed. For example, *tcsetattr()* is specified in place of the use of the TCSETA *ioctl()* command function. This allows specification of all the arguments in a manner consistent with the ISO C standard unlike the varying third argument of *ioctl()*, which is sometimes a pointer (to any of many different types) and sometimes an **int**.

The advantages of this new method include:

- It allows strict type checking.
- The direction of transfer of control data is explicit.
- Portable capabilities are clearly identified.
- The need for a general interface routine is avoided.
- Size of the argument is well-defined (there is only one type).

The disadvantages include:

- No historical implementation used the new method.
- There are many small routines instead of one general-purpose one.
- The historical parallel with *fcntl()* is broken.

The issue of modem control was excluded from IEEE Std 1003.1-2001 on the grounds that:

- It was concerned with setting and control of hardware timers.
- The appropriate timers and settings vary widely internationally.
- Feedback from European computer manufacturers indicated that this facility was not consistent with European needs and that specification of such a facility was not a requirement for portability.

## A.11.1 Interface Characteristics

### A.11.1.1 Opening a Terminal Device File

There is no additional rationale provided for this section.

### A.11.1.2 Process Groups

There is a potential race when the members of the foreground process group on a terminal leave that process group, either by exit or by changing process groups. After the last process exits the process group, but before the foreground process group ID of the terminal is changed (usually by a job control shell), it would be possible for a new process to be created with its process ID equal to the terminal's foreground process group ID. That process might then become the process group leader and accidentally be placed into the foreground on a terminal that was not necessarily its controlling terminal. As a result of this problem, the controlling terminal is defined to not have a foreground process group during this time.

The cases where a controlling terminal has no foreground process group occur when all processes in the foreground process group either terminate and are waited for or join other process groups via *setpgid()* or *setsid()*. If the process group leader terminates, this is the first case described; if it leaves the process group via *setpgid()*, this is the second case described (a process group leader cannot successfully call *setsid()*). When one of those cases causes a controlling terminal to have no foreground process group, it has two visible effects on applications. The first is the value returned by *tcgetpgrp()*. The second (which occurs only in the



case where the process group leader terminates) is the sending of signals in response to special input characters. The intent of IEEE Std 1003.1-2001 is that no process group be wrongly identified as the foreground process group by *tcgetpgrp()* or unintentionally receive signals because of placement into the foreground.

In 4.3 BSD, the old process group ID continues to be used to identify the foreground process group and is returned by the function equivalent to *tcgetpgrp()*. In that implementation it is possible for a newly created process to be assigned the same value as a process ID and then form a new process group with the same value as a process group ID. The result is that the new process group would receive signals from this terminal for no apparent reason, and IEEE Std 1003.1-2001 precludes this by forbidding a process group from entering the foreground in this way. It would be more direct to place part of the requirement made by the last sentence under *fork()*, but there is no convenient way for that section to refer to the value that *tcgetpgrp()* returns, since in this case there is no process group and thus no process group ID.

One possibility for a conforming implementation is to behave similarly to 4.3 BSD, but to prevent this reuse of the ID, probably in the implementation of *fork()*, as long as it is in use by the terminal.

Another possibility is to recognize when the last process stops using the terminal's foreground process group ID, which is when the process group lifetime ends, and to change the terminal's foreground process group ID to a reserved value that is never used as a process ID or process group ID. (See the definition of *process group lifetime* in the definitions section.) The process ID can then be reserved until the terminal has another foreground process group.

The 4.3 BSD implementation permits the leader (and only member) of the foreground process group to leave the process group by calling the equivalent of *setpgid()* and to later return, expecting to return to the foreground. There are no known application needs for this behavior, and IEEE Std 1003.1-2001 neither requires nor forbids it (except that it is forbidden for session leaders) by leaving it unspecified.

#### A.11.1.3 The Controlling Terminal

IEEE Std 1003.1-2001 does not specify a mechanism by which to allocate a controlling terminal. This is normally done by a system utility (such as *getty*) and is considered an administrative feature outside the scope of IEEE Std 1003.1-2001.

Historical implementations allocate controlling terminals on certain *open()* calls. Since *open()* is part of POSIX.1, its behavior had to be dealt with. The traditional behavior is not required because it is not very straightforward or flexible for either implementations or applications. However, because of its prevalence, it was not practical to disallow this behavior either. Thus, a mechanism was standardized to ensure portable, predictable behavior in *open()*.

Some historical implementations deallocate a controlling terminal on the last system-wide close. This behavior is neither required nor prohibited. Even on implementations that do provide this behavior, applications generally cannot depend on it due to its system-wide nature.

#### A.11.1.4 Terminal Access Control

The access controls described in this section apply only to a process that is accessing its controlling terminal. A process accessing a terminal that is not its controlling terminal is effectively treated the same as a member of the foreground process group. While this may seem unintuitive, note that these controls are for the purpose of job control, not security, and job control relates only to a process' controlling terminal. Normal file access permissions handle security.

2815 If the process calling *read()* or *write()* is in a background process group that is orphaned, it is not  
 2816 desirable to stop the process group, as it is no longer under the control of a job control shell that  
 2817 could put it into the foreground again. Accordingly, calls to *read()* or *write()* functions by such  
 2818 processes receive an immediate error return. This is different from 4.2 BSD, which kills orphaned  
 2819 processes that receive terminal stop signals.

2820 The foreground/background/orphaned process group check performed by the terminal driver  
 2821 must be repeatedly performed until the calling process moves into the foreground or until the  
 2822 process group of the calling process becomes orphaned. That is, when the terminal driver  
 2823 determines that the calling process is in the background and should receive a job control signal,  
 2824 it sends the appropriate signal (SIGTTIN or SIGTTOU) to every process in the process group of  
 2825 the calling process and then it allows the calling process to immediately receive the signal. The  
 2826 latter is typically performed by blocking the process so that the signal is immediately noticed.  
 2827 Note, however, that after the process finishes receiving the signal and control is returned to the  
 2828 driver, the terminal driver must re-execute the foreground/background/orphaned process  
 2829 group check. The process may still be in the background, either because it was continued in the  
 2830 background by a job control shell, or because it caught the signal and did nothing.

2831 The terminal driver repeatedly performs the foreground/background/orphaned process group  
 2832 checks whenever a process is about to access the terminal. In the case of *write()* or the control  
 2833 *tc\*()* functions, the check is performed at the entry of the function. In the case of *read()*, the check  
 2834 is performed not only at the entry of the function, but also after blocking the process to wait for  
 2835 input characters (if necessary). That is, once the driver has determined that the process calling  
 2836 the *read()* function is in the foreground, it attempts to retrieve characters from the input queue. If  
 2837 the queue is empty, it blocks the process waiting for characters. When characters are available  
 2838 and control is returned to the driver, the terminal driver must return to the repeated  
 2839 foreground/background/orphaned process group check again. The process may have moved  
 2840 from the foreground to the background while it was blocked waiting for input characters.

#### 2841 A.11.1.5 Input Processing and Reading Data

2842 There is no additional rationale provided for this section.

#### 2843 A.11.1.6 Canonical Mode Input Processing

2844 The term “character” is intended here. ERASE should erase the last character, not the last byte.  
 2845 In the case of multi-byte characters, these two may be different.

2846 4.3 BSD has a WERASE character that erases the last “word” typed (but not any preceding  
 2847 <blank>s or <tab>s). A word is defined as a sequence of non-<blank>s, with <tab>s counted as  
 2848 <blank>s. Like ERASE, WERASE does not erase beyond the beginning of the line. This  
 2849 WERASE feature has not been specified in POSIX.1 because it is difficult to define in the  
 2850 international environment. It is only useful for languages where words are delimited by  
 2851 <blank>s. In some ideographic languages, such as Japanese and Chinese, words are not  
 2852 delimited at all. The WERASE character should presumably go back to the beginning of a  
 2853 sentence in those cases; practically, this means it would not be used much for those languages.

2854 It should be noted that there is a possible inherent deadlock if the application and  
 2855 implementation conflict on the value of {MAX\_CANON}. With ICANON set (if IXOFF is  
 2856 enabled) and more than {MAX\_CANON} characters transmitted without a <linefeed>,  
 2857 transmission will be stopped, the <linefeed> (or <carriage-return> when ICRLF is set) will never  
 2858 arrive, and the *read()* will never be satisfied.

2859 An application should not set IXOFF if it is using canonical mode unless it knows that (even in  
 2860 the face of a transmission error) the conditions described previously cannot be met or unless it  
 2861 is prepared to deal with the possible deadlock in some other way, such as timeouts.

2862 It should also be noted that this can be made to happen in non-canonical mode if the trigger  
2863 value for sending IXOFF is less than VMIN and VTIME is zero.

#### 2864 A.11.1.7 Non-Canonical Mode Input Processing

2865 Some points to note about MIN and TIME:

- 2866 1. The interactions of MIN and TIME are not symmetric. For example, when MIN>0 and  
2867 TIME=0, TIME has no effect. However, in the opposite case where MIN=0 and TIME>0,  
2868 both MIN and TIME play a role in that MIN is satisfied with the receipt of a single  
2869 character.
- 2870 2. Also note that in case A (MIN>0, TIME>0), TIME represents an inter-character timer, while  
2871 in case C (MIN=0, TIME>0), TIME represents a read timer.

2872 These two points highlight the dual purpose of the MIN/TIME feature. Cases A and B, where  
2873 MIN>0, exist to handle burst-mode activity (for example, file transfer programs) where a  
2874 program would like to process at least MIN characters at a time. In case A, the inter-character  
2875 timer is activated by a user as a safety measure; in case B, it is turned off.

2876 Cases C and D exist to handle single-character timed transfers. These cases are readily adaptable  
2877 to screen-based applications that need to know if a character is present in the input queue before  
2878 refreshing the screen. In case C, the read is timed; in case D, it is not.

2879 Another important note is that MIN is always just a minimum. It does not denote a record  
2880 length. That is, if a program does a read of 20 bytes, MIN is 10, and 25 characters are present, 20  
2881 characters are returned to the user. In the special case of MIN=0, this still applies: if more than  
2882 one character is available, they all will be returned immediately.

#### 2883 A.11.1.8 Writing Data and Output Processing

2884 There is no additional rationale provided for this section.

#### 2885 A.11.1.9 Special Characters

2886 There is no additional rationale provided for this section.

#### 2887 A.11.1.10 Modem Disconnect

2888 There is no additional rationale provided for this section.

#### 2889 A.11.1.11 Closing a Terminal Device File

2890 IEEE Std 1003.1-2001 does not specify that a *close()* on a terminal device file include the  
2891 equivalent of a call to *tcflow(fd,TCOON)*.

2892 An implementation that discards output at the time *close()* is called after reporting the return  
2893 value to the *write()* call that data was written does not conform with IEEE Std 1003.1-2001. An  
2894 application has functions such as *tcdrain()*, *tcflush()*, and *tcflow()* available to obtain the detailed  
2895 behavior it requires with respect to flushing of output.

2896 At the time of the last close on a terminal device, an application relinquishes any ability to exert  
2897 flow control via *tcflow()*.

## 2898 A.11.2 Parameters that Can be Set

### 2899 A.11.2.1 The *termios* Structure

2900 This structure is part of an interface that, in general, retains the historic grouping of flags.  
 2901 Although a more optimal structure for implementations may be possible, the degree of change  
 2902 to applications would be significantly larger.

### 2903 A.11.2.2 Input Modes

2904 Some historical implementations treated a long break as multiple events, as many as one per  
 2905 character time. The wording in POSIX.1 explicitly prohibits this.

2906 Although the ISTRIP flag is normally superfluous with today's terminal hardware and software,  
 2907 it is historically supported. Therefore, applications may be using ISTRIP, and there is no  
 2908 technical problem with supporting this flag. Also, applications may wish to receive only 7-bit  
 2909 input bytes and may not be connected directly to the hardware terminal device (for example,  
 2910 when a connection traverses a network).

2911 Also, there is no requirement in general that the terminal device ensures that high-order bits  
 2912 beyond the specified character size are cleared. ISTRIP provides this function for 7-bit  
 2913 characters, which are common.

2914 In dealing with multi-byte characters, the consequences of a parity error in such a character, or in  
 2915 an escape sequence affecting the current character set, are beyond the scope of POSIX.1 and are  
 2916 best dealt with by the application processing the multi-byte characters.

### 2917 A.11.2.3 Output Modes

2918 POSIX.1 does not describe postprocessing of output to a terminal or detailed control of that from  
 2919 a conforming application. (That is, translation of <newline> to <carriage-return> followed by  
 2920 <linefeed> or <tab> processing.) There is nothing that a conforming application should do to its  
 2921 output for a terminal because that would require knowledge of the operation of the terminal. It  
 2922 is the responsibility of the operating system to provide postprocessing appropriate to the output  
 2923 device, whether it is a terminal or some other type of device.

2924 Extensions to POSIX.1 to control the type of postprocessing already exist and are expected to  
 2925 continue into the future. The control of these features is primarily to adjust the interface between  
 2926 the system and the terminal device so the output appears on the display correctly. This should  
 2927 be set up before use by any application.

2928 In general, both the input and output modes should not be set absolutely, but rather modified  
 2929 from the inherited state.

### 2930 A.11.2.4 Control Modes

2931 This section could be misread that the symbol "CSIZE" is a title in the **termios** *c\_cflag* field.  
 2932 Although it does serve that function, it is also a required symbol, as a literal reading of POSIX.1  
 2933 (and the caveats about typography) would indicate.

### 2934 A.11.2.5 Local Modes

2935 Non-canonical mode is provided to allow fast bursts of input to be read efficiently while still  
 2936 allowing single-character input.

2937 The ECHONL function historically has been in many implementations. Since there seems to be  
 2938 no technical problem with supporting ECHONL, it is included in POSIX.1 to increase consensus.

2939 The alternate behavior possible when ECHOK or ECHOE are specified with ICANON is  
 2940 permitted as a compromise depending on what the actual terminal hardware can do. Erasing  
 2941 characters and lines is preferred, but is not always possible.

#### 2942 A.11.2.6 Special Control Characters

2943 Permitting VMIN and VTIME to overlap with VEOF and VEOL was a compromise for historical  
 2944 implementations. Only when backwards-compatibility of object code is a serious concern to an  
 2945 implementor should an implementation continue this practice. Correct applications that work  
 2946 with the overlap (at the source level) should also work if it is not present, but not the reverse.

## 2947 A.12 Utility Conventions

### 2948 A.12.1 Utility Argument Syntax

2949 The standard developers considered that recent trends toward diluting the SYNOPSIS sections  
 2950 of historical reference pages to the equivalent of:

2951 `command [options][operands]`

2952 were a disservice to the reader. Therefore, considerable effort was placed into rigorous  
 2953 definitions of all the command line arguments and their interrelationships. The relationships  
 2954 depicted in the synopses are normative parts of IEEE Std 1003.1-2001; this information is  
 2955 sometimes repeated in textual form, but that is only for clarity within context.

2956 The use of “undefined” for conflicting argument usage and for repeated usage of the same  
 2957 option is meant to prevent conforming applications from using conflicting arguments or  
 2958 repeated options unless specifically allowed (as is the case with *ls*, which allows simultaneous,  
 2959 repeated use of the *-C*, *-l*, and *-l* options). Many historical implementations will tolerate this  
 2960 usage, choosing either the first or the last applicable argument. This tolerance can continue, but  
 2961 conforming applications cannot rely upon it. (Other implementations may choose to print usage  
 2962 messages instead.)

2963 The use of “undefined” for conflicting argument usage also allows an implementation to make  
 2964 reasonable extensions to utilities where the implementor considers mutually-exclusive options  
 2965 according to IEEE Std 1003.1-2001 to have a sensible meaning and result.

2966 IEEE Std 1003.1-2001 does not define the result of a command when an option-argument or  
 2967 operand is not followed by ellipses and the application specifies more than one of that option-  
 2968 argument or operand. This allows an implementation to define valid (although non-standard)  
 2969 behavior for the utility when more than one such option or operand is specified.

2970 The following table summarizes the requirements for option-arguments:

	SYNOPSIS Shows:		
	<i>-a arg</i>	<i>-barg</i>	<i>-c[arg]</i>
Conforming application uses:	<i>-a arg</i>	<i>-barg</i>	<i>-carg</i> or <i>-c</i>
System supports:	<i>-a arg</i> and <i>-aarg</i>	<i>-b arg</i> and <i>-barg</i>	<i>-carg</i> and <i>-c</i>
Non-conforming applications may use:	<i>-aarg</i>	<i>-b arg</i>	N/A

2978 Allowing <blank>s after an option (that is, placing an option and its option-argument into  
 2979 separate argument strings) when IEEE Std 1003.1-2001 does not require it encourages portability  
 2980 of users, while still preserving backwards-compatibility of scripts. Inserting <blank>s between

the option and the option-argument is preferred; however, historical usage has not been consistent in this area; therefore, <blank>s are required to be handled by all implementations, but implementations are also allowed to handle the historical syntax. Another justification for selecting the multiple-argument method was that the single-argument case is inherently ambiguous when the option-argument can legitimately be a null string.

IEEE Std 1003.1-2001 explicitly states that digits are permitted as operands and option-arguments. The lower and upper bounds for the values of the numbers used for operands and option-arguments were derived from the ISO C standard values for {LONG\_MIN} and {LONG\_MAX}. The requirement on the standard utilities is that numbers in the specified range do not cause a syntax error, although the specification of a number need not be semantically correct for a particular operand or option-argument of a utility. For example, the specification of:

```
dd obs=3000000000
```

would yield undefined behavior for the application and could be a syntax error because the number 3 000 000 000 is outside of the range -2 147 483 647 to +2 147 483 647. On the other hand:

```
dd obs=2000000000
```

may cause some error, such as “blocksize too large”, rather than a syntax error.

## A.12.2 Utility Syntax Guidelines

This section is based on the rules listed in the SVID. It was included for two reasons:

1. The individual utility descriptions in the Shell and Utilities volume of IEEE Std 1003.1-2001, Chapter 4, Utilities needed a set of common (although not universal) actions on which they could anchor their descriptions of option and operand syntax. Most of the standard utilities actually do use these guidelines, and many of their historical implementations use the *getopt()* function for their parsing. Therefore, it was simpler to cite the rules and merely identify exceptions.
2. Writers of conforming applications need suggested guidelines if the POSIX community is to avoid the chaos of historical UNIX system command syntax.

It is recommended that all *future* utilities and applications use these guidelines to enhance “user portability”. The fact that some historical utilities could not be changed (to avoid breaking historical applications) should not deter this future goal.

The voluntary nature of the guidelines is highlighted by repeated uses of the word *should* throughout. This usage should not be misinterpreted to imply that utilities that claim conformance in their OPTIONS sections do not always conform.

Guidelines 1 and 2 encourage utility writers to use only characters from the portable character set because use of locale-specific characters may make the utility inaccessible from other locales. Use of uppercase letters is discouraged due to problems associated with porting utilities to systems that do not distinguish between uppercase and lowercase characters in filenames. Use of non-alphanumeric characters is discouraged due to the number of utilities that treat non-alphanumeric characters in “special” ways depending on context (such as the shell using whitespace characters to delimit arguments, various quote characters for quoting, the dollar sign to introduce variable expansion, etc.).

In the Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.9.1, Simple Commands, it is further stated that a command used in the Shell Command Language cannot be named with a trailing colon.

Guideline 3 was changed to allow alphanumeric characters (letters and digits) from the character set to allow compatibility with historical usage. Historical practice allows the use of digits

wherever practical, and there are no portability issues that would prohibit the use of digits. In fact, from an internationalization viewpoint, digits (being non-language-dependent) are preferable over letters (a `-2` is intuitively self-explanatory to any user, while in the `-f filename` the letter `'f'` is a mnemonic aid only to speakers of Latin-based languages where “filename” happens to translate to a word that begins with `'f'`). Since guideline 3 still retains the word “single”, multi-digit options are not allowed. Instances of historical utilities that used them have been marked obsolescent, with the numbers being changed from option names to option-arguments.

It was difficult to achieve a satisfactory solution to the problem of name space in option characters. When the standard developers desired to extend the historical `cc` utility to accept ISO C standard programs, they found that all of the portable alphabet was already in use by various vendors. Thus, they had to devise a new name, `c89` (now superseded by `c99`), rather than something like `cc -X`. There were suggestions that implementors be restricted to providing extensions through various means (such as using a plus sign as the option delimiter or using option characters outside the alphanumeric set) that would reserve all of the remaining alphanumeric characters for future POSIX standards. These approaches were resisted because they lacked the historical style of UNIX systems. Furthermore, if a vendor-provided option should become commonly used in the industry, it would be a candidate for standardization. It would be desirable to standardize such a feature using historical practice for the syntax (the semantics can be standardized with any syntax). This would not be possible if the syntax was one reserved for the vendor. However, since the standardization process may lead to minor changes in the semantics, it may prove to be better for a vendor to use a syntax that will not be affected by standardization.

Guideline 8 includes the concept of comma-separated lists in a single argument. It is up to the utility to parse such a list itself because `getopt()` just returns the single string. This situation was retained so that certain historical utilities would not violate the guidelines. Applications preparing for international use should be aware of an occasional problem with comma-separated lists: in some locales, the comma is used as the radix character. Thus, if an application is preparing operands for a utility that expects a comma-separated list, it should avoid generating non-integer values through one of the means that is influenced by setting the `LC_NUMERIC` variable (such as `awk`, `bc`, `printf`, or `printf()`).

Applications calling any utility with a first operand starting with `'--'` should usually specify `--`, as indicated by Guideline 10, to mark the end of the options. This is true even if the SYNOPSIS in the Shell and Utilities volume of IEEE Std 1003.1-2001 does not specify any options; implementations may provide options as extensions to the Shell and Utilities volume of IEEE Std 1003.1-2001. The standard utilities that do not support Guideline 10 indicate that fact in the OPTIONS section of the utility description.

Guideline 11 was modified to clarify that the order of different options should not matter relative to one another. However, the order of repeated options that also have option-arguments may be significant; therefore, such options are required to be interpreted in the order that they are specified. The `make` utility is an instance of a historical utility that uses repeated options in which the order is significant. Multiple files are specified by giving multiple instances of the `-f` option; for example:

```
make -f common_header -f specific_rules target
```

Guideline 13 does not imply that all of the standard utilities automatically accept the operand `'--'` to mean standard input or output, nor does it specify the actions of the utility upon encountering multiple `'--'` operands. It simply says that, by default, `'--'` operands are not used for other purposes in the file reading or writing (but not when using `stat()`, `unlink()`, `touch`, and so on) utilities. All information concerning actual treatment of the `'--'` operand is found in the

3075 individual utility sections.

3076 An area of concern was that as implementations mature, implementation-defined utilities and  
3077 implementation-defined utility options will result. The idea was expressed that there needed to  
3078 be a standard way, say an environment variable or some such mechanism, to identify  
3079 implementation-defined utilities separately from standard utilities that may have the same  
3080 name. It was decided that there already exist several ways of dealing with this situation and that  
3081 it is outside of the scope to attempt to standardize in the area of non-standard items. A method  
3082 that exists on some historical implementations is the use of the so-called `/local/bin` or  
3083 `/usr/local/bin` directory to separate local or additional copies or versions of utilities. Another  
3084 method that is also used is to isolate utilities into completely separate domains. Still another  
3085 method to ensure that the desired utility is being used is to request the utility by its full  
3086 pathname. There are many approaches to this situation; the examples given above serve to  
3087 illustrate that there is more than one.

## 3088 **A.13 Headers**

### 3089 **A.13.1 Format of Entries**

3090 Each header reference page has a common layout of sections describing the interface. This layout  
3091 is similar to the manual page or “man” page format shipped with most UNIX systems, and each  
3092 header has sections describing the SYNOPSIS and DESCRIPTION. These are the two sections  
3093 that relate to conformance.

3094 Additional sections are informative, and add considerable information for the application  
3095 developer. APPLICATION USAGE sections provide additional caveats, issues, and  
3096 recommendations to the developer. RATIONALE sections give additional information on the  
3097 decisions made in defining the interface.

3098 FUTURE DIRECTIONS sections act as pointers to related work that may impact the interface in  
3099 the future, and often cautions the developer to architect the code to account for a change in this  
3100 area. Note that a future directions statement should not be taken as a commitment to adopt a  
3101 feature or interface in the future.

3102 The CHANGE HISTORY section describes when the interface was introduced, and how it has  
3103 changed.

3104 Option labels and margin markings in the page can be useful in guiding the application  
3105 developer.



3106 / *Rationale (Informative)*

3107 **Part B:**

3108 **System Interfaces**

3109 *The Open Group*

3110 *The Institute of Electrical and Electronics Engineers, Inc.*



# Rationale for System Interfaces

3111

## 3112 **B.1 Introduction**

### 3113 **B.1.1 Scope**

3114 Refer to Section A.1.1 (on page 3).

### 3115 **B.1.2 Conformance**

3116 Refer to Section A.2 (on page 9).

### 3117 **B.1.3 Normative References**

3118 There is no additional rationale provided for this section.

### 3119 **B.1.4 Change History**

3120 The change history is provided as an informative section, to track changes from previous issues  
3121 of IEEE Std 1003.1-2001.

3122 The following sections describe changes made to the System Interfaces volume of  
3123 IEEE Std 1003.1-2001 since Issue 5 of the base document. The CHANGE HISTORY section for  
3124 each entry details the technical changes that have been made to that entry from Issue 5. Changes  
3125 between earlier issues of the base document and Issue 5 are not included.

3126 The change history between Issue 5 and Issue 6 also lists the changes since the  
3127 ISO POSIX-1: 1996 standard.

#### 3128 **Changes from Issue 5 to Issue 6 (IEEE Std 1003.1-2001)**

3129 The following list summarizes the major changes that were made in the System Interfaces  
3130 volume of IEEE Std 1003.1-2001 from Issue 5 to Issue 6:

- 3131 • This volume of IEEE Std 1003.1-2001 is extensively revised so that it can be both an IEEE  
3132 POSIX Standard and an Open Group Technical Standard.
- 3133 • The POSIX System Interfaces requirements incorporate support of FIPS 151-2.
- 3134 • The POSIX System Interfaces requirements are updated to align with some features of the  
3135 Single UNIX Specification.
- 3136 • A RATIONALE section is added to each reference page.
- 3137 • Networking interfaces from the XNS, Issue 5.2 specification are incorporated.
- 3138 • IEEE Std 1003.1d-1999 is incorporated.
- 3139 • IEEE Std 1003.1j-2000 is incorporated.
- 3140 • IEEE Std 1003.1q-2000 is incorporated.
- 3141 • IEEE P1003.1a draft standard is incorporated.

- Existing functionality is aligned with the ISO/IEC 9899: 1999 standard.
- New functionality from the ISO/IEC 9899: 1999 standard is incorporated.
- IEEE PASC Interpretations are applied.
- The Open Group corrigenda and resolutions are applied.

### New Features in Issue 6

The functions first introduced in Issue 6 (over the Issue 5 Base document) are listed in the table below:

New Functions in Issue 6		
<i>acosf()</i>	<i>catanhl()</i>	<i>cprojf()</i>
<i>acoshf()</i>	<i>catanl()</i>	<i>cprojl()</i>
<i>acoshl()</i>	<i>cbrtf()</i>	<i>creal()</i>
<i>acosl()</i>	<i>cbrtl()</i>	<i>crealf()</i>
<i>asinf()</i>	<i>ccos()</i>	<i>creall()</i>
<i>asinhf()</i>	<i>ccosf()</i>	<i>csin()</i>
<i>asinhl()</i>	<i>ccosh()</i>	<i>csinf()</i>
<i>asinl()</i>	<i>ccoshf()</i>	<i>csinh()</i>
<i>atan2f()</i>	<i>ccoshl()</i>	<i>csinhf()</i>
<i>atan2l()</i>	<i>ccosl()</i>	<i>csinhl()</i>
<i>atanf()</i>	<i>ceilf()</i>	<i>csinl()</i>
<i>atanhf()</i>	<i>ceil()</i>	<i>csqrt()</i>
<i>atanhl()</i>	<i>cexp()</i>	<i>csqrtf()</i>
<i>atanl()</i>	<i>cexpf()</i>	<i>csqrtl()</i>
<i>atoll()</i>	<i>cexpl()</i>	<i>ctan()</i>
<i>cabs()</i>	<i>cimag()</i>	<i>ctanf()</i>
<i>cabsf()</i>	<i>cimagf()</i>	<i>ctanh()</i>
<i>cabsl()</i>	<i>cimagl()</i>	<i>ctanhf()</i>
<i>cacos()</i>	<i>clock_getcpuclockid()</i>	<i>ctanhl()</i>
<i>cacosf()</i>	<i>clock_nanosleep()</i>	<i>ctanl()</i>
<i>cacosh()</i>	<i>clog()</i>	<i>erfcf()</i>
<i>cacoshf()</i>	<i>clogf()</i>	<i>erfcl()</i>
<i>cacoshl()</i>	<i>clogl()</i>	<i>erff()</i>
<i>cacosl()</i>	<i>conj()</i>	<i>erfl()</i>
<i>carg()</i>	<i>conjf()</i>	<i>exp2()</i>
<i>cargf()</i>	<i>conjl()</i>	<i>exp2f()</i>
<i>cargl()</i>	<i>copysign()</i>	<i>exp2l()</i>
<i>casin()</i>	<i>copysignf()</i>	<i>expf()</i>
<i>casinf()</i>	<i>copysignl()</i>	<i>expl()</i>
<i>casinh()</i>	<i>cosf()</i>	<i>expm1f()</i>
<i>casinhf()</i>	<i>coshf()</i>	<i>expm1l()</i>
<i>casinhl()</i>	<i>coshl()</i>	<i>fabsf()</i>
<i>casinl()</i>	<i>cosl()</i>	<i>fabsl()</i>
<i>catan()</i>	<i>cpow()</i>	<i>fdim()</i>
<i>catanf()</i>	<i>cpowf()</i>	<i>fdimf()</i>
<i>catanh()</i>	<i>cpowl()</i>	<i>fdiml()</i>
<i>catanhf()</i>	<i>cproj()</i>	<i>feclearexcept()</i>

3188  
3189  
3190  
3191  
3192  
3193  
3194  
3195  
3196  
3197  
3198  
3199  
3200  
3201  
3202  
3203  
3204  
3205  
3206  
3207  
3208  
3209  
3210  
3211  
3212  
3213  
3214  
3215  
3216  
3217  
3218  
3219  
3220  
3221  
3222  
3223  
3224  
3225  
3226  
3227  
3228  
3229  
3230  
3231  
3232  
3233

### New Functions in Issue 6

<i>fegetenv()</i>	<i>ldexpl()</i>	<i>posix_fallocate()</i>
<i>fegetexceptflag()</i>	<i>lgammaf()</i>	<i>posix_madvise()</i>
<i>fegetround()</i>	<i>lgammal()</i>	<i>posix_mem_offset()</i>
<i>feholdexcept()</i>	<i>llabs()</i>	<i>posix_memalign()</i>
<i>feraiseexcept()</i>	<i>lldiv()</i>	<i>posix_openpt()</i>
<i>fesetenv()</i>	<i>llrint()</i>	<i>posix_spawn()</i>
<i>fesetexceptflag()</i>	<i>llrintf()</i>	<i>posix_spawn_file_actions_addclose()</i>
<i>fesetround()</i>	<i>llrintl()</i>	<i>posix_spawn_file_actions_adddup2()</i>
<i>fetestexcept()</i>	<i>llround()</i>	<i>posix_spawn_file_actions_addopen()</i>
<i>feupdateenv()</i>	<i>llroundf()</i>	<i>posix_spawn_file_actions_destroy()</i>
<i>floorf()</i>	<i>llroundl()</i>	<i>posix_spawn_file_actions_init()</i>
<i>floorl()</i>	<i>log10f()</i>	<i>posix_spawnattr_destroy()</i>
<i>fma()</i>	<i>log10l()</i>	<i>posix_spawnattr_getflags()</i>
<i>fnaf()</i>	<i>log1pf()</i>	<i>posix_spawnattr_getpgroup()</i>
<i>fnal()</i>	<i>log1pl()</i>	<i>posix_spawnattr_getschedparam()</i>
<i>fmax()</i>	<i>log2()</i>	<i>posix_spawnattr_getschedpolicy()</i>
<i>fmaxf()</i>	<i>log2f()</i>	<i>posix_spawnattr_getsigdefault()</i>
<i>fmaxl()</i>	<i>log2l()</i>	<i>posix_spawnattr_getsigmask()</i>
<i>fmin()</i>	<i>logbf()</i>	<i>posix_spawnattr_init()</i>
<i>fminf()</i>	<i>logbl()</i>	<i>posix_spawnattr_setflags()</i>
<i>fminl()</i>	<i>logf()</i>	<i>posix_spawnattr_setpgroup()</i>
<i>fmodf()</i>	<i>logl()</i>	<i>posix_spawnattr_setschedparam()</i>
<i>fmodl()</i>	<i>lrint()</i>	<i>posix_spawnattr_setschedpolicy()</i>
<i>fpclassify()</i>	<i>lrintf()</i>	<i>posix_spawnattr_setsigdefault()</i>
<i>frexpf()</i>	<i>lrintl()</i>	<i>posix_spawnattr_setsigmask()</i>
<i>frexpl()</i>	<i>lround()</i>	<i>posix_spawnnp()</i>
<i>hypotf()</i>	<i>lroundf()</i>	<i>posix_trace_attr_destroy()</i>
<i>hypotl()</i>	<i>lroundl()</i>	<i>posix_trace_attr_getclockres()</i>
<i>ilogbf()</i>	<i>modff()</i>	<i>posix_trace_attr_getcreatetime()</i>
<i>ilogbl()</i>	<i>modfl()</i>	<i>posix_trace_attr_getgenversion()</i>
<i>imaxabs()</i>	<i>mq_timedreceive()</i>	<i>posix_trace_attr_getinherited()</i>
<i>imaxdiv()</i>	<i>mq_timedsend()</i>	<i>posix_trace_attr_getlogfullpolicy()</i>
<i>isblank()</i>	<i>nan()</i>	<i>posix_trace_attr_getlogsize()</i>
<i>isfinite()</i>	<i>nanf()</i>	<i>posix_trace_attr_getmaxdatasize()</i>
<i>isgreater()</i>	<i>nanl()</i>	<i>posix_trace_attr_getmaxsystemeventsizesize()</i>
<i>isgreaterequal()</i>	<i>nearbyint()</i>	<i>posix_trace_attr_getmaxuserereventsizesize()</i>
<i>isinf()</i>	<i>nearbyintf()</i>	<i>posix_trace_attr_getname()</i>
<i>isless()</i>	<i>nearbyintl()</i>	<i>posix_trace_attr_getstreamfullpolicy()</i>
<i>islessequal()</i>	<i>nextafterf()</i>	<i>posix_trace_attr_getstreamsize()</i>
<i>islessgreater()</i>	<i>nextafterl()</i>	<i>posix_trace_attr_init()</i>
<i>isnormal()</i>	<i>nexttoward()</i>	<i>posix_trace_attr_setinherited()</i>
<i>isunordered()</i>	<i>nexttowardf()</i>	<i>posix_trace_attr_setlogfullpolicy()</i>
<i>iswblank()</i>	<i>nexttowardl()</i>	<i>posix_trace_attr_setlogsize()</i>
<i>ldexpl()</i>	<i>posix_fadvise()</i>	<i>posix_trace_create()</i>

3234  
3235  
3236  
3237  
3238  
3239  
3240  
3241  
3242  
3243  
3244  
3245  
3246  
3247  
3248  
3249  
3250  
3251  
3252  
3253  
3254  
3255  
3256  
3257  
3258  
3259  
3260  
3261  
3262  
3263  
3264  
3265  
3266  
3267  
3268  
3269  
3270  
3271  
3272  
3273  
3274

### New Functions in Issue 6

<i>posix_trace_attr_setmaxdatasize()</i>	<i>pthread_barrier_destroy()</i>	<i>signbit()</i>
<i>posix_trace_attr_setname()</i>	<i>pthread_barrier_init()</i>	<i>sinf()</i>
<i>posix_trace_attr_setstreamfullpolicy()</i>	<i>pthread_barrier_wait()</i>	<i>sinhf()</i>
<i>posix_trace_attr_setstreamsize()</i>	<i>pthread_barrierattr_destroy()</i>	<i>sinhl()</i>
<i>posix_trace_clear()</i>	<i>pthread_barrierattr_getpshared()</i>	<i>sinl()</i>
<i>posix_trace_close()</i>	<i>pthread_barrierattr_init()</i>	<i>socketmark()</i>
<i>posix_trace_create_withlog()</i>	<i>pthread_barrierattr_setpshared()</i>	<i>sqrtrf()</i>
<i>posix_trace_event()</i>	<i>pthread_condattr_getclock()</i>	<i>sqrtrl()</i>
<i>posix_trace_eventid_equal()</i>	<i>pthread_condattr_setclock()</i>	<i>strerror_r()</i>
<i>posix_trace_eventid_get_name()</i>	<i>pthread_getcpuclockid()</i>	<i>strtoimax()</i>
<i>posix_trace_eventid_open()</i>	<i>pthread_mutex_timedlock()</i>	<i>strtoll()</i>
<i>posix_trace_eventset_add()</i>	<i>pthread_rwlock_timedrdlock()</i>	<i>strtoull()</i>
<i>posix_trace_eventset_del()</i>	<i>pthread_rwlock_timedwrlock()</i>	<i>strtoumax()</i>
<i>posix_trace_eventset_empty()</i>	<i>pthread_setschedprio()</i>	<i>tanf()</i>
<i>posix_trace_eventset_fill()</i>	<i>pthread_spin_destroy()</i>	<i>tanhf()</i>
<i>posix_trace_eventset_ismember()</i>	<i>pthread_spin_init()</i>	<i>tanhrl()</i>
<i>posix_trace_eventtypelist_getnext_id()</i>	<i>pthread_spin_lock()</i>	<i>tanl()</i>
<i>posix_trace_eventtypelist_rewind()</i>	<i>pthread_spin_trylock()</i>	<i>tgamma()</i>
<i>posix_trace_flush()</i>	<i>pthread_spin_unlock()</i>	<i>tgammalf()</i>
<i>posix_trace_get_attr()</i>	<i>remainderf()</i>	<i>tgammal()</i>
<i>posix_trace_get_filter()</i>	<i>remainderl()</i>	<i>trunc()</i>
<i>posix_trace_get_status()</i>	<i>remquo()</i>	<i>truncf()</i>
<i>posix_trace_getnext_event()</i>	<i>remquof()</i>	<i>truncl()</i>
<i>posix_trace_open()</i>	<i>remquol()</i>	<i>unsetenv()</i>
<i>posix_trace_rewind()</i>	<i>rintf()</i>	<i>vfprintf()</i>
<i>posix_trace_set_filter()</i>	<i>rintl()</i>	<i>vfscanf()</i>
<i>posix_trace_shutdown()</i>	<i>round()</i>	<i>vfwscanf()</i>
<i>posix_trace_start()</i>	<i>roundf()</i>	<i>vprintf()</i>
<i>posix_trace_stop()</i>	<i>roundl()</i>	<i>vscanf()</i>
<i>posix_trace_timedgetnext_event()</i>	<i>scalbln()</i>	<i>vsprintf()</i>
<i>posix_trace_trid_eventid_open()</i>	<i>scalblnf()</i>	<i>vsscanf()</i>
<i>posix_trace_trygetnext_event()</i>	<i>scalblnl()</i>	<i>vswscanf()</i>
<i>posix_typed_mem_get_info()</i>	<i>scalbn()</i>	<i>vwscanf()</i>
<i>posix_typed_mem_open()</i>	<i>scalbnf()</i>	<i>wcstoimax()</i>
<i>powf()</i>	<i>scalbnl()</i>	<i>wcstoll()</i>
<i>powl()</i>	<i>sem_timedwait()</i>	<i>wcstoull()</i>
<i>pselect()</i>	<i>setgid()</i>	<i>wcstoumax()</i>
<i>pthread_attr_getstack()</i>	<i>setenv()</i>	
<i>pthread_attr_setstack()</i>	<i>setuid()</i>	

3275           The following new headers are introduced in Issue 6:

3276

3277

3278

3279

3280

New Headers in Issue 6		
<complex.h>	<spawn.h>	<tgmath.h>
<fenv.h>	<stdbool.h>	<trace.h>
<net/if.h>	<stdint.h>	

The following table lists the functions and symbols from the XSI extension. These are new since the ISO POSIX-1: 1996 standard.

New XSI Functions and Symbols in Issue 6			
<code>_longjmp()</code>	<code>getcontext()</code>	<code>msgget()</code>	<code>setstate()</code>
<code>_setjmp()</code>	<code>getdate()</code>	<code>msgrcv()</code>	<code>setutxent()</code>
<code>_tolower()</code>	<code>getgrent()</code>	<code>msgsnd()</code>	<code>shmat()</code>
<code>_toupper()</code>	<code>gethostid()</code>	<code>nftw()</code>	<code>shmctl()</code>
<code>a64l()</code>	<code>getitimer()</code>	<code>nice()</code>	<code>shmdt()</code>
<code>basename()</code>	<code>getpgid()</code>	<code>nl_langinfo()</code>	<code>shmget()</code>
<code>bcmp()</code>	<code>getpmsg()</code>	<code>nrnd48()</code>	<code>sigaltstack()</code>
<code>bcopy()</code>	<code>getpriority()</code>	<code>openlog()</code>	<code>sighold()</code>
<code>bzero()</code>	<code>getpwent()</code>	<code>poll()</code>	<code>sigignore()</code>
<code>catclose()</code>	<code>getrlimit()</code>	<code>posix_openpt()</code>	<code>siginterrupt()</code>
<code>catgets()</code>	<code>getrusage()</code>	<code>pread()</code>	<code>sigpause()</code>
<code>catopen()</code>	<code>getsid()</code>	<code>pthread_attr_getguardsize()</code>	<code>sigrelse()</code>
<code>closelog()</code>	<code>getsubopt()</code>	<code>pthread_attr_setguardsize()</code>	<code>sigset()</code>
<code>crypt()</code>	<code>gettimeofday()</code>	<code>pthread_attr_setstack()</code>	<code>srand48()</code>
<code>daylight</code>	<code>getutxent()</code>	<code>pthread_getconcurrency()</code>	<code>srandom()</code>
<code>dbm_clearerr()</code>	<code>getutxid()</code>	<code>pthread_mutexattr_gettype()</code>	<code>statvfs()</code>
<code>dbm_close()</code>	<code>getutxline()</code>	<code>pthread_mutexattr_settype()</code>	<code>strcasecmp()</code>
<code>dbm_delete()</code>	<code>getwd()</code>	<code>pthread_rwlockattr_init()</code>	<code>strdup()</code>
<code>dbm_error()</code>	<code>grantpt()</code>	<code>pthread_rwlockattr_setpshared()</code>	<code>strfmon()</code>
<code>dbm_fetch()</code>	<code>hcreate()</code>	<code>pthread_setconcurrency()</code>	<code>strncasecmp()</code>
<code>dbm_firstkey()</code>	<code>hdestroy()</code>	<code>ptsname()</code>	<code>strptime()</code>
<code>dbm_nextkey()</code>	<code>hsearch()</code>	<code>putenv()</code>	<code>swab()</code>
<code>dbm_open()</code>	<code>iconv()</code>	<code>pututxline()</code>	<code>swapcontext()</code>
<code>dbm_store()</code>	<code>iconv_close()</code>	<code>pwrite()</code>	<code>sync()</code>
<code>dirname()</code>	<code>iconv_open()</code>	<code>random()</code>	<code>syslog()</code>
<code>dlclose()</code>	<code>index()</code>	<code>readv()</code>	<code>tcgetsid()</code>
<code>dlderror()</code>	<code>initstate()</code>	<code>realpath()</code>	<code>tdelete()</code>
<code>dlopen()</code>	<code>insque()</code>	<code>remque()</code>	<code>telldir()</code>
<code>dlsym()</code>	<code>isascii()</code>	<code>rindex()</code>	<code>tempnam()</code>
<code>drand48()</code>	<code>jrand48()</code>	<code>seed48()</code>	<code>tfind()</code>
<code>ecvt()</code>	<code>killpg()</code>	<code>seekdir()</code>	<code>timezone</code>
<code>encrypt()</code>	<code>l64a()</code>	<code>semctl()</code>	<code>toascii()</code>
<code>endgrent()</code>	<code>lchown()</code>	<code>semget()</code>	<code>truncate()</code>
<code>endpwent()</code>	<code>lcong48()</code>	<code>semop()</code>	<code>tsearch()</code>
<code>endutxent()</code>	<code>lfind()</code>	<code>setcontext()</code>	<code>twalk()</code>
<code>erand48()</code>	<code>lockf()</code>	<code>setgrent()</code>	<code>ulimit()</code>
<code>fchdir()</code>	<code>lrand48()</code>	<code>setitimer()</code>	<code>unlockpt()</code>
<code>fcvt()</code>	<code>lsearch()</code>	<code>setkey()</code>	<code>utimes()</code>
<code>ffs()</code>	<code>makecontext()</code>	<code>setlogmask()</code>	<code>waitid()</code>
<code>fntmsg()</code>	<code>memccpy()</code>	<code>setpgid()</code>	<code>wcswcs()</code>
<code>fstatvfs()</code>	<code>mknod()</code>	<code>setpriority()</code>	<code>wcswidth()</code>
<code>ftime()</code>	<code>mkstemp()</code>	<code>setpwent()</code>	<code>wcwidth()</code>
<code>ftok()</code>	<code>mktemp()</code>	<code>setregid()</code>	<code>writev()</code>
<code>ftw()</code>	<code>mrnd48()</code>	<code>setreuid()</code>	
<code>gcvt()</code>	<code>msgctl()</code>	<code>setrlimit()</code>	



The following table lists the headers from the XSI extension. These are new since the ISO POSIX-1:1996 standard.

New XSI Headers in Issue 6		
<cpio.h>	<poll.h>	<sys/statvfs.h>
<dlfcn.h>	<search.h>	<sys/time.h>
<fmtmsg.h>	<strings.h>	<sys/timeb.h>
<ftw.h>	<stropts.h>	<sys/uio.h>
<iconv.h>	<sys/ipc.h>	<syslog.h>
<langinfo.h>	<sys/mman.h>	<ucontext.h>
<libgen.h>	<sys/msg.h>	<ulimit.h>
<monetary.h>	<sys/resource.h>	<utmpx.h>
<ndbm.h>	<sys/sem.h>	
<nl_types.h>	<sys/shm.h>	

### **B.1.5 Terminology**

Refer to Section A.1.4 (on page 5).

### **B.1.6 Definitions**

Refer to Section A.3 (on page 13).

### **B.1.7 Relationship to Other Formal Standards**

There is no additional rationale provided for this section.

### **B.1.8 Portability**

Refer to Section A.1.5 (on page 8).

#### **B.1.8.1 Codes**

Refer to Section A.1.5.1 (on page 8).

### **B.1.9 Format of Entries**

Each system interface reference page has a common layout of sections describing the interface. This layout is similar to the manual page or “man” page format shipped with most UNIX systems, and each header has sections describing the SYNOPSIS, DESCRIPTION, RETURN VALUE, and ERRORS. These are the four sections that relate to conformance.

Additional sections are informative, and add considerable information for the application developer. EXAMPLES sections provide example usage. APPLICATION USAGE sections provide additional caveats, issues, and recommendations to the developer. RATIONALE sections give additional information on the decisions made in defining the interface.

FUTURE DIRECTIONS sections act as pointers to related work that may impact the interface in the future, and often cautions the developer to architect the code to account for a change in this area. Note that a future directions statement should not be taken as a commitment to adopt a feature or interface in the future.

The CHANGE HISTORY section describes when the interface was introduced, and how it has changed.

Option labels and margin markings in the page can be useful in guiding the application developer.

## 3371 B.2 General Information

### 3372 B.2.1 Use and Implementation of Functions

The information concerning the use of functions was adapted from a description in the ISO C standard. Here is an example of how an application program can protect itself from functions that may or may not be macros, rather than true functions:

The *atoi()* function may be used in any of several ways:

- By use of its associated header (possibly generating a macro expansion):

```
3378     #include <stdlib.h>
3379     /* ... */
3380     i = atoi(str);
```

- By use of its associated header (assuredly generating a true function call):

```
3382     #include <stdlib.h>
3383     #undef atoi
3384     /* ... */
3385     i = atoi(str);
```

or:

```
3387     #include <stdlib.h>
3388     /* ... */
3389     i = (atoi) (str);
```

- By explicit declaration:

```
3391     extern int atoi (const char *);
3392     /* ... */
3393     i = atoi(str);
```

- By implicit declaration:

```
3395     /* ... */
3396     i = atoi(str);
```

(Assuming no function prototype is in scope. This is not allowed by the ISO C standard for functions with variable arguments; furthermore, parameter type conversion “widening” is subject to different rules in this case.)

Note that the ISO C standard reserves names starting with ‘\_’ for the compiler. Therefore, the compiler could, for example, implement an intrinsic, built-in function *\_asm\_builtin\_atoi()*, which it recognized and expanded into inline assembly code. Then, in *<stdlib.h>*, there could be the following:

```
3404     #define atoi(X) _asm_builtin_atoi(X)
```

The user’s “normal” call to *atoi()* would then be expanded inline, but the implementor would also be required to provide a callable function named *atoi()* for use when the application requires it; for example, if its address is to be stored in a function pointer variable.

## 3408 **B.2.2 The Compilation Environment**

### 3409 *B.2.2.1 POSIX.1 Symbols*

3410 This and the following section address the issue of “name space pollution”. The ISO C standard  
 3411 requires that the name space beyond what it reserves not be altered except by explicit action of  
 3412 the application writer. This section defines the actions to add the POSIX.1 symbols for those  
 3413 headers where both the ISO C standard and POSIX.1 need to define symbols, and also where the  
 3414 XSI extension extends the base standard.

3415 When headers are used to provide symbols, there is a potential for introducing symbols that the  
 3416 application writer cannot predict. Ideally, each header should only contain one set of symbols,  
 3417 but this is not practical for historical reasons. Thus, the concept of feature test macros is  
 3418 included. Two feature test macros are explicitly defined by IEEE Std 1003.1-2001; it is expected  
 3419 that future revisions may add to this.

3420 **Note:** Feature test macros allow an application to announce to the implementation its desire to have  
 3421 certain symbols and prototypes exposed. They should not be confused with the version test  
 3422 macros and constants for options in `<unistd.h>` which are the implementation’s way of  
 3423 announcing functionality to the application.

3424 It is further intended that these feature test macros apply only to the headers specified by  
 3425 IEEE Std 1003.1-2001. Implementations are expressly permitted to make visible symbols not  
 3426 specified by IEEE Std 1003.1-2001, within both POSIX.1 and other headers, under the control of  
 3427 feature test macros that are not defined by IEEE Std 1003.1-2001.

### 3428 **The `_POSIX_C_SOURCE` Feature Test Macro**

3429 Since `_POSIX_SOURCE` specified by the POSIX.1-1990 standard did not have a value associated  
 3430 with it, the `_POSIX_C_SOURCE` macro replaces it, allowing an application to inform the system  
 3431 of the revision of the standard to which it conforms. This symbol will allow implementations to  
 3432 support various revisions of IEEE Std 1003.1-2001 simultaneously. For instance, when either  
 3433 `_POSIX_SOURCE` is defined or `_POSIX_C_SOURCE` is defined as 1, the system should make  
 3434 visible the same name space as permitted and required by the POSIX.1-1990 standard. When  
 3435 `_POSIX_C_SOURCE` is defined, the state of `_POSIX_SOURCE` is completely irrelevant.

3436 It is expected that C bindings to future POSIX standards will define new values for  
 3437 `_POSIX_C_SOURCE`, with each new value reserving the name space for that new standard, plus  
 3438 all earlier POSIX standards.

### 3439 **The `_XOPEN_SOURCE` Feature Test Macro**

3440 The feature test macro `_XOPEN_SOURCE` is provided as the announcement mechanism for the  
 3441 application that it requires functionality from the Single UNIX Specification. `_XOPEN_SOURCE`  
 3442 must be defined to the value 600 before the inclusion of any header to enable the functionality in  
 3443 the Single UNIX Specification. Its definition subsumes the use of `_POSIX_SOURCE` and  
 3444 `_POSIX_C_SOURCE`.

3445 An extract of code from a conforming application, that appears before any **#include** statements,  
 3446 is given below:

```
3447 #define _XOPEN_SOURCE 600 /* Single UNIX Specification, Version 3 */
3448 #include ...
```

3449 Note that the definition of `_XOPEN_SOURCE` with the value 600 makes the definition of  
 3450 `_POSIX_C_SOURCE` redundant and it can safely be omitted.

## 3451 B.2.2.2 The Name Space

3452 The reservation of identifiers is paraphrased from the ISO C standard. The text is included  
 3453 because it needs to be part of IEEE Std 1003.1-2001, regardless of possible changes in future  
 3454 versions of the ISO C standard.

3455 These identifiers may be used by implementations, particularly for feature test macros.  
 3456 Implementations should not use feature test macro names that might be reasonably used by a  
 3457 standard.

3458 Including headers more than once is a reasonably common practice, and it should be carried  
 3459 forward from the ISO C standard. More significantly, having definitions in more than one  
 3460 header is explicitly permitted. Where the potential declaration is “benign” (the same definition  
 3461 twice) the declaration can be repeated, if that is permitted by the compiler. (This is usually true  
 3462 of macros, for example.) In those situations where a repetition is not benign (for example,  
 3463 **typedefs**), conditional compilation must be used. The situation actually occurs both within the  
 3464 ISO C standard and within POSIX.1: **time\_t** should be in **<sys/types.h>**, and the ISO C standard  
 3465 mandates that it be in **<time.h>**.

3466 The area of name space pollution *versus* additions to structures is difficult because of the macro  
 3467 structure of C. The following discussion summarizes all the various problems with and  
 3468 objections to the issue.

3469 Note the phrase “user-defined macro”. Users are not permitted to define macro names (or any  
 3470 other name) beginning with “\_**[A-Z\_]**”. Thus, the conflict cannot occur for symbols reserved  
 3471 to the vendor’s name space, and the permission to add fields automatically applies, without  
 3472 qualification, to those symbols.

3473 1. Data structures (and unions) need to be defined in headers by implementations to meet  
 3474 certain requirements of POSIX.1 and the ISO C standard.

3475 2. The structures defined by POSIX.1 are typically minimal, and any practical  
 3476 implementation would wish to add fields to these structures either to hold additional  
 3477 related information or for backwards-compatibility (or both). Future standards (and *de*  
 3478 *facto* standards) would also wish to add to these structures. Issues of field alignment make  
 3479 it impractical (at least in the general case) to simply omit fields when they are not defined  
 3480 by the particular standard involved.

3481 The **dirent** structure is an example of such a minimal structure (although one could argue  
 3482 about whether the other fields need visible names). The **st\_rdev** field of most  
 3483 implementations’ **stat** structure is a common example where extension is needed and  
 3484 where a conflict could occur.

3485 3. Fields in structures are in an independent name space, so the addition of such fields  
 3486 presents no problem to the C language itself in that such names cannot interact with  
 3487 identically named user symbols because access is qualified by the specific structure name.

3488 4. There is an exception to this: macro processing is done at a lexical level. Thus, symbols  
 3489 added to a structure might be recognized as user-provided macro names at the location  
 3490 where the structure is declared. This only can occur if the user-provided name is declared  
 3491 as a macro before the header declaring the structure is included. The user’s use of the name  
 3492 after the declaration cannot interfere with the structure because the symbol is hidden and  
 3493 only accessible through access to the structure. Presumably, the user would not declare  
 3494 such a macro if there was an intention to use that field name.

3495 5. Macros from the same or a related header might use the additional fields in the structure,  
 3496 and those field names might also collide with user macros. Although this is a less frequent  
 3497 occurrence, since macros are expanded at the point of use, no constraint on the order of use

of names can apply.

6. An “obvious” solution of using names in the reserved name space and then redefining them as macros when they should be visible does not work because this has the effect of exporting the symbol into the general name space. For example, given a (hypothetical) system-provided header **<h.h>**, and two parts of a C program in **a.c** and **b.c**, in header **<h.h>**:

```
struct foo {
    int __i;
}

#ifdef _FEATURE_TEST
#define i __i;
#endif
```

In file **a.c**:

```
#include h.h
extern int i;
...
```

In file **b.c**:

```
extern int i;
...
```

The symbol that the user thinks of as *i* in both files has an external name of `__i` in **a.c**; the same symbol *i* in **b.c** has an external name *i* (ignoring any hidden manipulations the compiler might perform on the names). This would cause a mysterious name resolution problem when **a.o** and **b.o** are linked.

Simply avoiding definition then causes alignment problems in the structure.

A structure of the form:

```
struct foo {
    union {
        int __i;
#ifdef _FEATURE_TEST
        int i;
#endif
    } __ii;
}
```

does not work because the name of the logical field *i* is `__ii.i`, and introduction of a macro to restore the logical name immediately reintroduces the problem discussed previously (although its manifestation might be more immediate because a syntax error would result if a recursive macro did not cause it to fail first).

1

7. A more workable solution would be to declare the structure:

```
struct foo {
#ifdef _FEATURE_TEST
    int i;
#else
    int __i;
#endif
}
```

However, if a macro (particularly one required by a standard) is to be defined that uses this field, two must be defined: one that uses *i*, the other that uses *\_\_i*. If more than one additional field is used in a macro and they are conditional on distinct combinations of features, the complexity goes up as  $2^n$ .

All this leaves a difficult situation: vendors must provide very complex headers to deal with what is conceptually simple and safe—adding a field to a structure. It is the possibility of user-provided macros with the same name that makes this difficult.

Several alternatives were proposed that involved constraining the user's access to part of the name space available to the user (as specified by the ISO C standard). In some cases, this was only until all the headers had been included. There were two proposals discussed that failed to achieve consensus:

1. Limiting it for the whole program.
2. Restricting the use of identifiers containing only uppercase letters until after all system headers had been included. It was also pointed out that because macros might wish to access fields of a structure (and macro expansion occurs totally at point of use) restricting names in this way would not protect the macro expansion, and thus the solution was inadequate.

It was finally decided that reservation of symbols would occur, but as constrained.

The current wording also allows the addition of fields to a structure, but requires that user macros of the same name not interfere. This allows vendors to do one of the following:

- Not create the situation (do not extend the structures with user-accessible names or use the solution in (7) above)
- Extend their compilers to allow some way of adding names to structures and macros safely

There are at least two ways that the compiler might be extended: add new preprocessor directives that turn off and on macro expansion for certain symbols (without changing the value of the macro) and a function or lexical operation that suppresses expansion of a word. The latter seems more flexible, particularly because it addresses the problem in macros as well as in declarations.

The following seems to be a possible implementation extension to the C language that will do this: any token that during macro expansion is found to be preceded by three '#' symbols shall not be further expanded in exactly the same way as described for macros that expand to their own name as in Section 3.8.3.4 of the ISO C standard. A vendor may also wish to implement this as an operation that is lexically a function, which might be implemented as:

```
#define __safe_name(x) ###x
```

Using a function notation would insulate vendors from changes in standards until such a functionality is standardized (if ever). Standardization of such a function would be valuable because it would then permit third parties to take advantage of it portably in software they may

3580	supply.	
3581	The symbols that are “explicitly permitted, but not required by IEEE Std 1003.1-2001” include	
3582	those classified below. (That is, the symbols classified below might, but are not required to, be	
3583	present when <code>_POSIX_C_SOURCE</code> is defined to have the value 200112L.)	
3584	• Symbols in <code>&lt;limits.h&gt;</code> and <code>&lt;unistd.h&gt;</code> that are defined to indicate support for options or	
3585	limits that are constant at compile-time	
3586	• Symbols in the name space reserved for the implementation by the ISO C standard	
3587	• Symbols in a name space reserved for a particular type of extension (for example, type names	
3588	ending with <code>_t</code> in <code>&lt;sys/types.h&gt;</code> )	
3589	• Additional members of structures or unions whose names do not reduce the name space	
3590	reserved for applications	
3591	Since both implementations and future revisions of IEEE Std 1003.1 and other POSIX standards	
3592	may use symbols in the reserved spaces described in these tables, there is a potential for name	
3593	space clashes. To avoid future name space clashes when adding symbols, implementations	
3594	should not use the <code>posix_</code> , <code>POSIX_</code> , or <code>_POSIX_</code> prefixes.	
3595	IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/2 is applied, deleting the entries <code>POSIX_</code> , <code>_POSIX_</code> , and <code>posix_</code> from the column of allowed name space prefixes for use by an implementation in the first table. The presence of these prefixes was contradicting later text which states that: “The prefixes <code>posix_</code> , <code>POSIX_</code> , and <code>_POSIX_</code> are reserved for use by Shell and Utilities volume of IEEE Std 1003.1-2001, Chapter 2, Shell Command Language and other POSIX standards. Implementations may add symbols to the headers shown in the following table, provided the identifiers . . . do not use the reserved prefixes <code>posix_</code> , <code>POSIX_</code> , or <code>_POSIX_</code> .”	1
3596		1
3597		1
3598		1
3599		1
3600		1
3601		1
3602	IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/3 is applied, correcting the reserved macro prefix from: “ <code>PRI[a-z]</code> , <code>SCN[a-z]</code> ” to: “ <code>PRI[Xa-z]</code> , <code>SCN[Xa-z]</code> ” in the second table. The change was needed since the ISO C standard allows implementations to define macros of the form <code>PRI</code> or <code>SCN</code> followed by any lowercase letter or ‘ <code>x</code> ’ in <code>&lt;inttypes.h&gt;</code> . (The ISO/IEC 9899:1999 standard, Subclause 7.26.4.)	1
3603		1
3604		1
3605		1
3606		1
3607	IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/4 is applied, adding a new section listing reserved names for the <code>&lt;stdint.h&gt;</code> header. This change is for alignment with the ISO C standard.	1
3608		1
3609	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/2 is applied, making it clear that implementations are permitted to have symbols with the prefix <code>_POSIX_</code> visible in any header.	2
3610		2
3611	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/3 is applied, updating the table of allowed macro prefixes to include the prefix <code>FP_[A-Z]</code> for <code>&lt;math.h&gt;</code> . This text is added for consistency with the <code>&lt;math.h&gt;</code> reference page in the Base Definitions volume of IEEE Std 1003.1-2001 which permits additional implementation-defined floating-point classifications.	2
3612		2
3613		2
3614		2
3615		2
3616	<b>B.2.3 Error Numbers</b>	
3617	It was the consensus of the standard developers that to allow the conformance document to state that an error occurs and under what conditions, but to disallow a statement that it never occurs, does not make sense. It could be implied by the current wording that this is allowed, but to reduce the possibility of future interpretation requests, it is better to make an explicit statement.	
3618		
3619		
3620		
3621		
3622	The ISO C standard requires that <code>errno</code> be an assignable lvalue. Originally, the definition in POSIX.1 was stricter than that in the ISO C standard, <code>extern int errno</code> , in order to support historical usage. In a multi-threaded environment, implementing <code>errno</code> as a global variable	
3623		
3624		

3625 results in non-deterministic results when accessed. It is required, however, that *errno* work as a  
 3626 per-thread error reporting mechanism. In order to do this, a separate *errno* value has to be  
 3627 maintained for each thread. The following section discusses the various alternative solutions  
 3628 that were considered.

3629 In order to avoid this problem altogether for new functions, these functions avoid using *errno*  
 3630 and, instead, return the error number directly as the function return value; a return value of zero  
 3631 indicates that no error was detected.

3632 For any function that can return errors, the function return value is not used for any purpose  
 3633 other than for reporting errors. Even when the output of the function is scalar, it is passed  
 3634 through a function argument. While it might have been possible to allow some scalar outputs to  
 3635 be coded as negative function return values and mixed in with positive error status returns, this  
 3636 was rejected—using the return value for a mixed purpose was judged to be of limited use and  
 3637 error prone.

3638 Checking the value of *errno* alone is not sufficient to determine the existence or type of an error,  
 3639 since it is not required that a successful function call clear *errno*. The variable *errno* should only  
 3640 be examined when the return value of a function indicates that the value of *errno* is meaningful.  
 3641 In that case, the function is required to set the variable to something other than zero.

3642 The variable *errno* is never set to zero by any function call; to do so would contradict the ISO C  
 3643 standard.

3644 POSIX.1 requires (in the ERRORS sections of function descriptions) certain error values to be set  
 3645 in certain conditions because many existing applications depend on them. Some error numbers,  
 3646 such as [EFAULT], are entirely implementation-defined and are noted as such in their  
 3647 description in the ERRORS section. This section otherwise allows wide latitude to the  
 3648 implementation in handling error reporting.

3649 Some of the ERRORS sections in IEEE Std 1003.1-2001 have two subsections. The first:

3650 “The function shall fail if:”

3651 could be called the “mandatory” section.

3652 The second:

3653 “The function may fail if:”

3654 could be informally known as the “optional” section.

3655 Attempting to infer the quality of an implementation based on whether it detects optional error  
 3656 conditions is not useful.

3657 Following each one-word symbolic name for an error, there is a description of the error. The  
 3658 rationale for some of the symbolic names follows:

3659 [ECANCELED] This spelling was chosen as being more common.

3660 [EFAULT] Most historical implementations do not catch an error and set *errno* when an  
 3661 invalid address is given to the functions *wait()*, *time()*, or *times()*. Some  
 3662 implementations cannot reliably detect an invalid address. And most systems  
 3663 that detect invalid addresses will do so only for a system call, not for a library  
 3664 routine.

3665 [EFTYPE] This error code was proposed in earlier proposals as “Inappropriate operation  
 3666 for file type”, meaning that the operation requested is not appropriate for the  
 3667 file specified in the function call. This code was proposed, although the same  
 3668 idea was covered by [ENOTTY], because the connotations of the name would



3669		be misleading. It was pointed out that the <i>fcntl()</i> function uses the error code
3670		[EINVAL] for this notion, and hence all instances of [EFTYPE] were changed
3671		to this code.
3672	[EINTR]	POSIX.1 prohibits conforming implementations from restarting interrupted
3673		system calls of conforming applications unless the SA_RESTART flag is in
3674		effect for the signal. However, it does not require that [EINTR] be returned
3675		when another legitimate value may be substituted; for example, a partial
3676		transfer count when <i>read()</i> or <i>write()</i> are interrupted. This is only given when
3677		the signal-catching function returns normally as opposed to returns by
3678		mechanisms like <i>longjmp()</i> or <i>siglongjmp()</i> .
3679	[ELOOP]	In specifying conditions under which implementations would generate this
3680		error, the following goals were considered:
3681		• To ensure that actual loops are detected, including loops that result from
3682		symbolic links across distributed file systems.
3683		• To ensure that during pathname resolution an application can rely on the
3684		ability to follow at least {SYMLOOP_MAX} symbolic links in the absence
3685		of a loop.
3686		• To allow implementations to provide the capability of traversing more
3687		than {SYMLOOP_MAX} symbolic links in the absence of a loop.
3688		• To allow implementations to detect loops and generate the error prior to
3689		encountering {SYMLOOP_MAX} symbolic links.
3690	[ENAMETOOLONG]	
3691		When a symbolic link is encountered during pathname resolution, the
3692		contents of that symbolic link are used to create a new pathname. The
3693		standard developers intended to allow, but not require, that implementations
3694		enforce the restriction of {PATH_MAX} on the result of this pathname
3695		substitution.
3696	[ENOMEM]	The term “main memory” is not used in POSIX.1 because it is
3697		implementation-defined.
3698	[ENOTSUP]	This error code is to be used when an implementation chooses to implement
3699		the required functionality of IEEE Std 1003.1-2001 but does not support
3700		optional facilities defined by IEEE Std 1003.1-2001. The return of [ENOSYS] is
3701		to be taken to indicate that the function of the interface is not supported at all;
3702		the function will always fail with this error code.
3703	[ENOTTY]	The symbolic name for this error is derived from a time when device control
3704		was done by <i>ioctl()</i> and that operation was only permitted on a terminal
3705		interface. The term “TTY” is derived from “teletypewriter”, the devices to
3706		which this error originally applied.
3707	[EOVERFLOW]	Most of the uses of this error code are related to large file support. Typically,
3708		these cases occur on systems which support multiple programming
3709		environments with different sizes for <b>off_t</b> , but they may also occur in
3710		connection with remote file systems.
3711		In addition, when different programming environments have different widths
3712		for types such as <b>int</b> and <b>uid_t</b> , several functions may encounter a condition
3713		where a value in a particular environment is too wide to be represented. In
3714		that case, this error should be raised. For example, suppose the currently

running process has 64-bit **int**, and file descriptor 9 223 372 036 854 775 807 is open and does not have the close-on-exec flag set. If the process then uses *execl()* to *exec* a file compiled in a programming environment with 32-bit **int**, the call to *execl()* can fail with *errno* set to [EOVERFLOW]. A similar failure can occur with *execl()* if any of the user IDs or any of the group IDs to be assigned to the new process image are out of range for the executed file's programming environment.

Note, however, that this condition cannot occur for functions that are explicitly described as always being successful, such as *getpid()*.

[EPIPE] This condition normally generates the signal SIGPIPE; the error is returned if the signal does not terminate the process.

[EROFS] In historical implementations, attempting to *unlink()* or *rmdir()* a mount point would generate an [EBUSY] error. An implementation could be envisioned where such an operation could be performed without error. In this case, if *either* the directory entry or the actual data structures reside on a read-only file system, [EROFS] is the appropriate error to generate. (For example, changing the link count of a file on a read-only file system could not be done, as is required by *unlink()*, and thus an error should be reported.)

Three error numbers, [EDOM], [EILSEQ], and [ERANGE], were added to this section primarily for consistency with the ISO C standard.

### Alternative Solutions for Per-Thread *errno*

The usual implementation of *errno* as a single global variable does not work in a multi-threaded environment. In such an environment, a thread may make a POSIX.1 call and get a *-1* error return, but before that thread can check the value of *errno*, another thread might have made a second POSIX.1 call that also set *errno*. This behavior is unacceptable in robust programs. There were a number of alternatives that were considered for handling the *errno* problem:

- Implement *errno* as a per-thread integer variable.
- Implement *errno* as a service that can access the per-thread error number.
- Change all POSIX.1 calls to accept an extra status argument and avoid setting *errno*.
- Change all POSIX.1 calls to raise a language exception.

The first option offers the highest level of compatibility with existing practice but requires special support in the linker, compiler, and/or virtual memory system to support the new concept of thread private variables. When compared with current practice, the third and fourth options are much cleaner, more efficient, and encourage a more robust programming style, but they require new versions of all of the POSIX.1 functions that might detect an error. The second option offers compatibility with existing code that uses the *<errno.h>* header to define the symbol *errno*. In this option, *errno* may be a macro defined:

```
#define errno  (*__errno())
extern int      *__errno();
```

This option may be implemented as a per-thread variable whereby an *errno* field is allocated in the user space object representing a thread, and whereby the function *\_\_errno()* makes a system call to determine the location of its user space object and returns the address of the *errno* field of that object. Another implementation, one that avoids calling the kernel, involves allocating stacks in chunks. The stack allocator keeps a side table indexed by chunk number containing a pointer to the thread object that uses that chunk. The *\_\_errno()* function then looks at the stack

3760 pointer, determines the chunk number, and uses that as an index into the chunk table to find its  
 3761 thread object and thus its private value of *errno*. On most architectures, this can be done in four  
 3762 to five instructions. Some compilers may wish to implement *\_\_errno()* inline to improve  
 3763 performance.

#### 3764 **Disallowing Return of the [EINTR] Error Code**

3765 Many blocking interfaces defined by IEEE Std 1003.1-2001 may return [EINTR] if interrupted  
 3766 during their execution by a signal handler. Blocking interfaces introduced under the Threads  
 3767 option do not have this property. Instead, they require that the interface appear to be atomic  
 3768 with respect to interruption. In particular, clients of blocking interfaces need not handle any  
 3769 possible [EINTR] return as a special case since it will never occur. If it is necessary to restart  
 3770 operations or complete incomplete operations following the execution of a signal handler, this is  
 3771 handled by the implementation, rather than by the application.

3772 Requiring applications to handle [EINTR] errors on blocking interfaces has been shown to be a  
 3773 frequent source of often unreproducible bugs, and it adds no compelling value to the available  
 3774 functionality. Thus, blocking interfaces introduced for use by multi-threaded programs do not  
 3775 use this paradigm. In particular, in none of the functions *flockfile()*, *pthread\_cond\_timedwait()*,  
 3776 *pthread\_cond\_wait()*, *pthread\_join()*, *pthread\_mutex\_lock()*, and *sigwait()* did providing [EINTR]  
 3777 returns add value, or even particularly make sense. Thus, these functions do not provide for an  
 3778 [EINTR] return, even when interrupted by a signal handler. The same arguments can be applied  
 3779 to *sem\_wait()*, *sem\_trywait()*, *sigwaitinfo()*, and *sigtimedwait()*, but implementations are  
 3780 permitted to return [EINTR] error codes for these functions for compatibility with earlier  
 3781 versions of IEEE Std 1003.1. Applications cannot rely on calls to these functions returning  
 3782 [EINTR] error codes when signals are delivered to the calling thread, but they should allow for  
 3783 the possibility.

#### 3784 **B.2.3.1 Additional Error Numbers**

3785 The ISO C standard defines the name space for implementations to add additional error  
 3786 numbers.

#### 3787 **B.2.4 Signal Concepts**

3788 Historical implementations of signals, using the *signal()* function, have shortcomings that make  
 3789 them unreliable for many application uses. Because of this, a new signal mechanism, based very  
 3790 closely on the one of 4.2 BSD and 4.3 BSD, was added to POSIX.1.

#### 3791 **Signal Names**

3792 The restriction on the actual type used for **sigset\_t** is intended to guarantee that these objects can  
 3793 always be assigned, have their address taken, and be passed as parameters by value. It is not  
 3794 intended that this type be a structure including pointers to other data structures, as that could  
 3795 impact the portability of applications performing such operations. A reasonable implementation  
 3796 could be a structure containing an array of some integer type.

3797 The signals described in IEEE Std 1003.1-2001 must have unique values so that they may be  
 3798 named as parameters of **case** statements in the body of a C-language **switch** clause. However,  
 3799 implementation-defined signals may have values that overlap with each other or with signals  
 3800 specified in IEEE Std 1003.1-2001. An example of this is SIGABRT, which traditionally overlaps  
 3801 some other signal, such as SIGIOT.

3802 SIGKILL, SIGTERM, SIGUSR1, and SIGUSR2 are ordinarily generated only through the explicit  
 3803 use of the *kill()* function, although some implementations generate SIGKILL under  
 3804 extraordinary circumstances. SIGTERM is traditionally the default signal sent by the *kill*

command.

The signals SIGBUS, SIGEMT, SIGIOT, SIGTRAP, and SIGSYS were omitted from POSIX.1 because their behavior is implementation-defined and could not be adequately categorized. Conforming implementations may deliver these signals, but must document the circumstances under which they are delivered and note any restrictions concerning their delivery. The signals SIGFPE, SIGILL, and SIGSEGV are similar in that they also generally result only from programming errors. They were included in POSIX.1 because they do indicate three relatively well-categorized conditions. They are all defined by the ISO C standard and thus would have to be defined by any system with an ISO C standard binding, even if not explicitly included in POSIX.1.

There is very little that a Conforming POSIX.1 Application can do by catching, ignoring, or masking any of the signals SIGILL, SIGTRAP, SIGIOT, SIGEMT, SIGBUS, SIGSEGV, SIGSYS, or SIGFPE. They will generally be generated by the system only in cases of programming errors. While it may be desirable for some robust code (for example, a library routine) to be able to detect and recover from programming errors in other code, these signals are not nearly sufficient for that purpose. One portable use that does exist for these signals is that a command interpreter can recognize them as the cause of a process' termination (with *wait()*) and print an appropriate message. The mnemonic tags for these signals are derived from their PDP-11 origin.

The signals SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU, and SIGCONT are provided for job control and are unchanged from 4.2 BSD. The signal SIGCHLD is also typically used by job control shells to detect children that have terminated or, as in 4.2 BSD, stopped.

Some implementations, including System V, have a signal named SIGCLD, which is similar to SIGCHLD in 4.2 BSD. POSIX.1 permits implementations to have a single signal with both names. POSIX.1 carefully specifies ways in which conforming applications can avoid the semantic differences between the two different implementations. The name SIGCHLD was chosen for POSIX.1 because most current application usages of it can remain unchanged in conforming applications. SIGCLD in System V has more cases of semantics that POSIX.1 does not specify, and thus applications using it are more likely to require changes in addition to the name change.

The signals SIGUSR1 and SIGUSR2 are commonly used by applications for notification of exceptional behavior and are described as "reserved as application-defined" so that such use is not prohibited. Implementations should not generate SIGUSR1 or SIGUSR2, except when explicitly requested by *kill()*. It is recommended that libraries not use these two signals, as such use in libraries could interfere with their use by applications calling the libraries. If such use is unavoidable, it should be documented. It is prudent for non-portable libraries to use non-standard signals to avoid conflicts with use of standard signals by portable libraries.

There is no portable way for an application to catch or ignore non-standard signals. Some implementations define the range of signal numbers, so applications can install signal-catching functions for all of them. Unfortunately, implementation-defined signals often cause problems when caught or ignored by applications that do not understand the reason for the signal. While the desire exists for an application to be more robust by handling all possible signals (even those only generated by *kill()*), no existing mechanism was found to be sufficiently portable to include in POSIX.1. The value of such a mechanism, if included, would be diminished given that SIGKILL would still not be catchable.

A number of new signal numbers are reserved for applications because the two user signals defined by POSIX.1 are insufficient for many realtime applications. A range of signal numbers is specified, rather than an enumeration of additional reserved signal names, because different applications and application profiles will require a different number of application signals. It is not desirable to burden all application domains and therefore all implementations with the

maximum number of signals required by all possible applications. Note that in this context, signal numbers are essentially different signal priorities.

The relatively small number of required additional signals, `{_POSIX_RTSIG_MAX}`, was chosen so as not to require an unreasonably large signal mask/set. While this number of signals defined in POSIX.1 will fit in a single 32-bit word signal mask, it is recognized that most existing implementations define many more signals than are specified in POSIX.1 and, in fact, many implementations have already exceeded 32 signals (including the “null signal”). Support of `{_POSIX_RTSIG_MAX}` additional signals may push some implementation over the single 32-bit word line, but is unlikely to push any implementations that are already over that line beyond the 64-signal line.

#### B.2.4.1 Signal Generation and Delivery

The terms defined in this section are not used consistently in documentation of historical systems. Each signal can be considered to have a lifetime beginning with generation and ending with delivery or acceptance. The POSIX.1 definition of “delivery” does not exclude ignored signals; this is considered a more consistent definition. This revised text in several parts of IEEE Std 1003.1-2001 clarifies the distinct semantics of asynchronous signal delivery and synchronous signal acceptance. The previous wording attempted to categorize both under the term “delivery”, which led to conflicts over whether the effects of asynchronous signal delivery applied to synchronous signal acceptance.

Signals generated for a process are delivered to only one thread. Thus, if more than one thread is eligible to receive a signal, one has to be chosen. The choice of threads is left entirely up to the implementation both to allow the widest possible range of conforming implementations and to give implementations the freedom to deliver the signal to the “easiest possible” thread should there be differences in ease of delivery between different threads.

Note that should multiple delivery among cooperating threads be required by an application, this can be trivially constructed out of the provided single-delivery semantics. The construction of a `sigwait_multiple()` function that accomplishes this goal is presented with the rationale for `sigwaitinfo()`.

Implementations should deliver unblocked signals as soon after they are generated as possible. However, it is difficult for POSIX.1 to make specific requirements about this, beyond those in `kill()` and `sigprocmask()`. Even on systems with prompt delivery, scheduling of higher priority processes is always likely to cause delays.

In general, the interval between the generation and delivery of unblocked signals cannot be detected by an application. Thus, references to pending signals generally apply to blocked, pending signals. An implementation registers a signal as pending on the process when no thread has the signal unblocked and there are no threads blocked in a `sigwait()` function for that signal. Thereafter, the implementation delivers the signal to the first thread that unblocks the signal or calls a `sigwait()` function on a signal set containing this signal rather than choosing the recipient thread at the time the signal is sent.

In the 4.3 BSD system, signals that are blocked and set to `SIG_IGN` are discarded immediately upon generation. For a signal that is ignored as its default action, if the action is `SIG_DFL` and the signal is blocked, a generated signal remains pending. In the 4.1 BSD system and in System V Release 3 (two other implementations that support a somewhat similar signal mechanism), all ignored blocked signals remain pending if generated. Because it is not normally useful for an application to simultaneously ignore and block the same signal, it was unnecessary for POSIX.1 to specify behavior that would invalidate any of the historical implementations.

There is one case in some historical implementations where an unblocked, pending signal does not remain pending until it is delivered. In the System V implementation of *signal()*, pending signals are discarded when the action is set to SIG\_DFL or a signal-catching routine (as well as to SIG\_IGN). Except in the case of setting SIGCHLD to SIG\_DFL, implementations that do this do not conform completely to POSIX.1. Some earlier proposals for POSIX.1 explicitly stated this, but these statements were redundant due to the requirement that functions defined by POSIX.1 not change attributes of processes defined by POSIX.1 except as explicitly stated.

POSIX.1 specifically states that the order in which multiple, simultaneously pending signals are delivered is unspecified. This order has not been explicitly specified in historical implementations, but has remained quite consistent and been known to those familiar with the implementations. Thus, there have been cases where applications (usually system utilities) have been written with explicit or implicit dependencies on this order. Implementors and others porting existing applications may need to be aware of such dependencies.

When there are multiple pending signals that are not blocked, implementations should arrange for the delivery of all signals at once, if possible. Some implementations stack calls to all pending signal-catching routines, making it appear that each signal-catcher was interrupted by the next signal. In this case, the implementation should ensure that this stacking of signals does not violate the semantics of the signal masks established by *sigaction()*. Other implementations process at most one signal when the operating system is entered, with remaining signals saved for later delivery. Although this practice is widespread, this behavior is neither standardized nor endorsed. In either case, implementations should attempt to deliver signals associated with the current state of the process (for example, SIGFPE) before other signals, if possible.

In 4.2 BSD and 4.3 BSD, it is not permissible to ignore or explicitly block SIGCONT, because if blocking or ignoring this signal prevented it from continuing a stopped process, such a process could never be continued (only killed by SIGKILL). However, 4.2 BSD and 4.3 BSD do block SIGCONT during execution of its signal-catching function when it is caught, creating exactly this problem. A proposal was considered to disallow catching SIGCONT in addition to ignoring and blocking it, but this limitation led to objections. The consensus was to require that SIGCONT always continue a stopped process when generated. This removed the need to disallow ignoring or explicit blocking of the signal; note that SIG\_IGN and SIG\_DFL are equivalent for SIGCONT.

#### B.2.4.2 Realtime Signal Generation and Delivery

The Realtime Signals Extension option to POSIX.1 signal generation and delivery behavior is required for the following reasons:

- The **sigevent** structure is used by other POSIX.1 functions that result in asynchronous event notifications to specify the notification mechanism to use and other information needed by the notification mechanism. IEEE Std 1003.1-2001 defines only three symbolic values for the notification mechanism:
  - SIGEV\_NONE is used to indicate that no notification is required when the event occurs. This is useful for applications that use asynchronous I/O with polling for completion.
  - SIGEV\_SIGNAL indicates that a signal is generated when the event occurs.
  - SIGEV\_THREAD provides for “callback functions” for asynchronous notifications done by a function call within the context of a new thread. This provides a multi-threaded process with a more natural means of notification than signals.

The primary difficulty with previous notification approaches has been to specify the environment of the notification routine.

- One approach is to limit the notification routine to call only functions permitted in a signal handler. While the list of permissible functions is clearly stated, this is overly restrictive.
- A second approach is to define a new list of functions or classes of functions that are explicitly permitted or not permitted. This would give a programmer more lists to deal with, which would be awkward.
- The third approach is to define completely the environment for execution of the notification function. A clear definition of an execution environment for notification is provided by executing the notification function in the environment of a newly created thread.

Implementations may support additional notification mechanisms by defining new values for *sigev\_notify*.

For a notification type of SIGEV\_SIGNAL, the other members of the **sigevent** structure defined by IEEE Std 1003.1-2001 specify the realtime signal—that is, the signal number and application-defined value that differentiates between occurrences of signals with the same number—that will be generated when the event occurs. The structure is defined in *<signal.h>*, even though the structure is not directly used by any of the signal functions, because it is part of the signals interface used by the POSIX.1b “client functions”. When the client functions include *<signal.h>* to define the signal names, the **sigevent** structure will also be defined.

An application-defined value passed to the signal handler is used to differentiate between different “events” instead of requiring that the application use different signal numbers for several reasons:

- Realtime applications potentially handle a very large number of different events. Requiring that implementations support a correspondingly large number of distinct signal numbers will adversely impact the performance of signal delivery because the signal masks to be manipulated on entry and exit to the handlers will become large.
- Event notifications are prioritized by signal number (the rationale for this is explained in the following paragraphs) and the use of different signal numbers to differentiate between the different event notifications overloads the signal number more than has already been done. It also requires that the application writer make arbitrary assignments of priority to events that are logically of equal priority.

A union is defined for the application-defined value so that either an integer constant or a pointer can be portably passed to the signal-catching function. On some architectures a pointer cannot be cast to an **int** and *vice versa*.

Use of a structure here with an explicit notification type discriminant rather than explicit parameters to realtime functions, or embedded in other realtime structures, provides for future extensions to IEEE Std 1003.1-2001. Additional, perhaps more efficient, notification mechanisms can be supported for existing realtime function interfaces, such as timers and asynchronous I/O, by extending the **sigevent** structure appropriately. The existing realtime function interfaces will not have to be modified to use any such new notification mechanism. The revised text concerning the SIGEV\_SIGNAL value makes consistent the semantics of the members of the **sigevent** structure, particularly in the definitions of *lio\_listio()* and *aio\_fsync()*. For uniformity, other revisions cause this specification to be referred to rather than inaccurately duplicated in the descriptions of functions and structures using the **sigevent** structure. The revised wording does not relax the requirement that the signal number be in the range SIGRTMIN to SIGRTMAX to guarantee queuing and passing of the application value, since that requirement is still implied by the signal names.

• IEEE Std 1003.1-2001 is intentionally vague on whether “non-realtime” signal-generating mechanisms can result in a **siginfo\_t** being supplied to the handler on delivery. In one existing implementation, a **siginfo\_t** is posted on signal generation, even though the implementation does not support queuing of multiple occurrences of a signal. It is not the intent of IEEE Std 1003.1-2001 to preclude this, independent of the mandate to define signals that do support queuing. Any interpretation that appears to preclude this is a mistake in the reading or writing of the standard.

• Signals handled by realtime signal handlers might be generated by functions or conditions that do not allow the specification of an application-defined value and do not queue. IEEE Std 1003.1-2001 specifies the *si\_code* member of the **siginfo\_t** structure used in existing practice and defines additional codes so that applications can detect whether an application-defined value is present or not. The code SI\_USER for *kill()*-generated signals is adopted from existing practice.

• The *sigaction()* *sa\_flags* value SA\_SIGINFO tells the implementation that the signal-catching function expects two additional arguments. When the flag is not set, a single argument, the signal number, is passed as specified by IEEE Std 1003.1-2001. Although IEEE Std 1003.1-2001 does not explicitly allow the *info* argument to the handler function to be NULL, this is existing practice. This provides for compatibility with programs whose signal-catching functions are not prepared to accept the additional arguments. IEEE Std 1003.1-2001 is explicitly unspecified as to whether signals actually queue when SA\_SIGINFO is not set for a signal, as there appear to be no benefits to applications in specifying one behavior or another. One existing implementation queues a **siginfo\_t** on each signal generation, unless the signal is already pending, in which case the implementation discards the new **siginfo\_t**; that is, the queue length is never greater than one. This implementation only examines SA\_SIGINFO on signal delivery, discarding the queued **siginfo\_t** if its delivery was not requested.

IEEE Std 1003.1-2001 specifies several new values for the *si\_code* member of the **siginfo\_t** structure. In existing practice, a *si\_code* value of less than or equal to zero indicates that the signal was generated by a process via the *kill()* function. In existing practice, values of *si\_code* that provide additional information for implementation-generated signals, such as SIGFPE or SIGSEGV, are all positive. Thus, if implementations define the new constants specified in IEEE Std 1003.1-2001 to be negative numbers, programs written to use existing practice will not break. IEEE Std 1003.1-2001 chose not to attempt to specify existing practice values of *si\_code* other than SI\_USER both because it was deemed beyond the scope of IEEE Std 1003.1-2001 and because many of the values in existing practice appear to be platform and implementation-defined. But, IEEE Std 1003.1-2001 does specify that if an implementation—for example, one that does not have existing practice in this area—chooses to define additional values for *si\_code*, these values have to be different from the values of the symbols specified by IEEE Std 1003.1-2001. This will allow conforming applications to differentiate between signals generated by one of the POSIX.1b asynchronous events and those generated by other implementation events in a manner compatible with existing practice.

The unique values of *si\_code* for the POSIX.1b asynchronous events have implications for implementations of, for example, asynchronous I/O or message passing in user space library code. Such an implementation will be required to provide a hidden interface to the signal generation mechanism that allows the library to specify the standard values of *si\_code*.

Existing practice also defines additional members of **siginfo\_t**, such as the process ID and user ID of the sending process for *kill()*-generated signals. These members were deemed not necessary to meet the requirements of realtime applications and are not specified by IEEE Std 1003.1-2001. Neither are they precluded.



The third argument to the signal-catching function, *context*, is left undefined by IEEE Std 1003.1-2001, but is specified in the interface because it matches existing practice for the SA\_SIGINFO flag. It was considered undesirable to require a separate implementation for SA\_SIGINFO for POSIX conformance on implementations that already support the two additional parameters.

- The requirement to deliver lower numbered signals in the range SIGRTMIN to SIGRTMAX first, when multiple unblocked signals are pending, results from several considerations:

- A method is required to prioritize event notifications. The signal number was chosen instead of, for instance, associating a separate priority with each request, because an implementation has to check pending signals at various points and select one for delivery when more than one is pending. Specifying a selection order is the minimal additional semantic that will achieve prioritized delivery. If a separate priority were to be associated with queued signals, it would be necessary for an implementation to search all non-empty, non-blocked signal queues and select from among them the pending signal with the highest priority. This would significantly increase the cost of and decrease the determinism of signal delivery.

- Given the specified selection of the lowest numeric unblocked pending signal, preemptive priority signal delivery can be achieved using signal numbers and signal masks by ensuring that the *sa\_mask* for each signal number blocks all signals with a higher numeric value.

For realtime applications that want to use only the newly defined realtime signal numbers without interference from the standard signals, this can be achieved by blocking all of the standard signals in the thread signal mask and in the *sa\_mask* installed by the signal action for the realtime signal handlers.

IEEE Std 1003.1-2001 explicitly leaves unspecified the ordering of signals outside of the range of realtime signals and the ordering of signals within this range with respect to those outside the range. It was believed that this would unduly constrain implementations or standards in the future definition of new signals.

#### B.2.4.3 Signal Actions

Early proposals mentioned SIGCONT as a second exception to the rule that signals are not delivered to stopped processes until continued. Because IEEE Std 1003.1-2001 now specifies that SIGCONT causes the stopped process to continue when it is generated, delivery of SIGCONT is not prevented because a process is stopped, even without an explicit exception to this rule.

Ignoring a signal by setting the action to SIG\_IGN (or SIG\_DFL for signals whose default action is to ignore) is not the same as installing a signal-catching function that simply returns. Invoking such a function will interrupt certain system functions that block processes (for example, *wait()*, *sigsuspend()*, *pause()*, *read()*, *write()*) while ignoring a signal has no such effect on the process.

Historical implementations discard pending signals when the action is set to SIG\_IGN. However, they do not always do the same when the action is set to SIG\_DFL and the default action is to ignore the signal. IEEE Std 1003.1-2001 requires this for the sake of consistency and also for completeness, since the only signal this applies to is SIGCHLD, and IEEE Std 1003.1-2001 disallows setting its action to SIG\_IGN.

Some implementations (System V, for example) assign different semantics for SIGCHLD depending on whether the action is set to SIG\_IGN or SIG\_DFL. Since POSIX.1 requires that the default action for SIGCHLD be to ignore the signal, applications should always set the action to SIG\_DFL in order to avoid SIGCHLD.

Whether or not an implementation allows SIG\_IGN as a SIGCHLD disposition to be inherited across a call to one of the *exec* family of functions or *posix\_spawn()* is explicitly left as unspecified. This change was made as a result of IEEE PASC Interpretation 1003.1 #132, and permits the implementation to decide between the following alternatives:

- Unconditionally leave SIGCHLD set to SIG\_IGN, in which case the implementation would not allow applications that assume inheritance of SIG\_DFL to conform to IEEE Std 1003.1-2001 without change. The implementation would, however, retain an ability to control applications that create child processes but never call on the *wait* family of functions, potentially filling up the process table.
- Unconditionally reset SIGCHLD to SIG\_DFL, in which case the implementation would allow applications that assume inheritance of SIG\_DFL to conform. The implementation would, however, lose an ability to control applications that spawn child processes but never reap them.
- Provide some mechanism, not specified in IEEE Std 1003.1-2001, to control inherited SIGCHLD dispositions.

Some implementations (System V, for example) will deliver a SIGCLD signal immediately when a process establishes a signal-catching function for SIGCLD when that process has a child that has already terminated. Other implementations, such as 4.3 BSD, do not generate a new SIGCHLD signal in this way. In general, a process should not attempt to alter the signal action for the SIGCHLD signal while it has any outstanding children. However, it is not always possible for a process to avoid this; for example, shells sometimes start up processes in pipelines with other processes from the pipeline as children. Processes that cannot ensure that they have no children when altering the signal action for SIGCHLD thus need to be prepared for, but not depend on, generation of an immediate SIGCHLD signal.

The default action of the stop signals (SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU) is to stop a process that is executing. If a stop signal is delivered to a process that is already stopped, it has no effect. In fact, if a stop signal is generated for a stopped process whose signal mask blocks the signal, the signal will never be delivered to the process since the process must receive a SIGCONT, which discards all pending stop signals, in order to continue executing.

The SIGCONT signal continues a stopped process even if SIGCONT is blocked (or ignored). However, if a signal-catching routine has been established for SIGCONT, it will not be entered until SIGCONT is unblocked.

If a process in an orphaned process group stops, it is no longer under the control of a job control shell and hence would not normally ever be continued. Because of this, orphaned processes that receive terminal-related stop signals (SIGTSTP, SIGTTIN, SIGTTOU, but not SIGSTOP) must not be allowed to stop. The goal is to prevent stopped processes from languishing forever. (As SIGSTOP is sent only via *kill()*, it is assumed that the process or user sending a SIGSTOP can send a SIGCONT when desired.) Instead, the system must discard the stop signal. As an extension, it may also deliver another signal in its place. 4.3 BSD sends a SIGKILL, which is overly effective because SIGKILL is not catchable. Another possible choice is SIGHUP. 4.3 BSD also does this for orphaned processes (processes whose parent has terminated) rather than for members of orphaned process groups; this is less desirable because job control shells manage process groups. POSIX.1 also prevents SIGTTIN and SIGTTOU signals from being generated for processes in orphaned process groups as a direct result of activity on a terminal, preventing infinite loops when *read()* and *write()* calls generate signals that are discarded; see Section A.11.1.4 (on page 69). A similar restriction on the generation of SIGTSTP was considered, but that would be unnecessary and more difficult to implement due to its asynchronous nature.

Although POSIX.1 requires that signal-catching functions be called with only one argument, there is nothing to prevent conforming implementations from extending POSIX.1 to pass additional arguments, as long as Strictly Conforming POSIX.1 Applications continue to compile and execute correctly. Most historical implementations do, in fact, pass additional, signal-specific arguments to certain signal-catching routines.

There was a proposal to change the declared type of the signal handler to:

```
void func (int sig, ...);
```

The usage of ellipses ("...") is ISO C standard syntax to indicate a variable number of arguments. Its use was intended to allow the implementation to pass additional information to the signal handler in a standard manner.

Unfortunately, this construct would require all signal handlers to be defined with this syntax because the ISO C standard allows implementations to use a different parameter passing mechanism for variable parameter lists than for non-variable parameter lists. Thus, all existing signal handlers in all existing applications would have to be changed to use the variable syntax in order to be standard and portable. This is in conflict with the goal of Minimal Changes to Existing Application Code.

When terminating a process from a signal-catching function, processes should be aware of any interpretation that their parent may make of the status returned by *wait()* or *waitpid()*. In particular, a signal-catching function should not call *exit(0)* or *\_exit(0)* unless it wants to indicate successful termination. A non-zero argument to *exit()* or *\_exit()* can be used to indicate unsuccessful termination. Alternatively, the process can use *kill()* to send itself a fatal signal (first ensuring that the signal is set to the default action and not blocked). See also the RATIONALE section of the *\_exit()* function.

The behavior of *unsafe* functions, as defined by this section, is undefined when they are invoked from signal-catching functions in certain circumstances. The behavior of reentrant functions, as defined by this section, is as specified by POSIX.1, regardless of invocation from a signal-catching function. This is the only intended meaning of the statement that reentrant functions may be used in signal-catching functions without restriction. Applications must still consider all effects of such functions on such things as data structures, files, and process state. In particular, application writers need to consider the restrictions on interactions when interrupting *sleep()* (see *sleep()*) and interactions among multiple handles for a file description. The fact that any specific function is listed as reentrant does not necessarily mean that invocation of that function from a signal-catching function is recommended.

In order to prevent errors arising from interrupting non-reentrant function calls, applications should protect calls to these functions either by blocking the appropriate signals or through the use of some programmatic semaphore. POSIX.1 does not address the more general problem of synchronizing access to shared data structures. Note in particular that even the "safe" functions may modify the global variable *errno*; the signal-catching function may want to save and restore its value. The same principles apply to the reentrancy of application routines and asynchronous data access.

Note that *longjmp()* and *siglongjmp()* are not in the list of reentrant functions. This is because the code executing after *longjmp()* or *siglongjmp()* can call any unsafe functions with the same danger as calling those unsafe functions directly from the signal handler. Applications that use *longjmp()* or *siglongjmp()* out of signal handlers require rigorous protection in order to be portable. Many of the other functions that are excluded from the list are traditionally implemented using either the C language *malloc()* or *free()* functions or the ISO C standard I/O library, both of which traditionally use data structures in a non-reentrant manner. Because any combination of different functions using a common data structure can cause reentrancy

4184 problems, POSIX.1 does not define the behavior when any unsafe function is called in a signal  
 4185 handler that interrupts any unsafe function.

4186 The only realtime extension to signal actions is the addition of the additional parameters to the  
 4187 signal-catching function. This extension has been explained and motivated in the previous  
 4188 section. In making this extension, though, developers of POSIX.1b ran into issues relating to  
 4189 function prototypes. In response to input from the POSIX.1 standard developers, members were  
 4190 added to the **sigaction** structure to specify function prototypes for the newer signal-catching  
 4191 function specified by POSIX.1b. These members follow changes that are being made to POSIX.1.  
 4192 Note that IEEE Std 1003.1-2001 explicitly states that these fields may overlap so that a union can  
 4193 be defined. This enabled existing implementations of POSIX.1 to maintain binary-compatibility  
 4194 when these extensions were added.

4195 The **siginfo\_t** structure was adopted for passing the application-defined value to match existing  
 4196 practice, but the existing practice has no provision for an application-defined value, so this was  
 4197 added. Note that POSIX normally reserves the “\_t” type designation for opaque types. The  
 4198 **siginfo\_t** structure breaks with this convention to follow existing practice and thus promote  
 4199 portability. Standardization of the existing practice for the other members of this structure may  
 4200 be addressed in the future.

4201 Although it is not explicitly visible to applications, there are additional semantics for signal  
 4202 actions implied by queued signals and their interaction with other POSIX.1b realtime functions.  
 4203 Specifically:

- 4204 • It is not necessary to queue signals whose action is SIG\_IGN.
- 4205 • For implementations that support POSIX.1b timers, some interaction with the timer functions  
 4206 at signal delivery is implied to manage the timer overrun count.

4207 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/5 is applied, reordering the RTS shaded 1  
 4208 text under the third and fourth paragraphs of the SIG\_DFL description. This corrects an earlier 1  
 4209 editorial error in this section. 1

4210 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/6 is applied, adding the *abort()* function 1  
 4211 to the list of async-cancel-safe functions. 1

4212 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/4 is applied, adding the *socketatmark()* 2  
 4213 function to the list of functions that shall be either reentrant or non-interruptible by signals and 2  
 4214 shall be async-signal-safe. 2

#### 4215 B.2.4.4 Signal Effects on Other Functions

4216 The most common behavior of an interrupted function after a signal-catching function returns is  
 4217 for the interrupted function to give an [EINTR] error unless the SA\_RESTART flag is in effect for  
 4218 the signal. However, there are a number of specific exceptions, including *sleep()* and certain  
 4219 situations with *read()* and *write()*.

4220 The historical implementations of many functions defined by IEEE Std 1003.1-2001 are not  
 4221 interruptible, but delay delivery of signals generated during their execution until after they  
 4222 complete. This is never a problem for functions that are guaranteed to complete in a short  
 4223 (imperceptible to a human) period of time. It is normally those functions that can suspend a  
 4224 process indefinitely or for long periods of time (for example, *wait()*, *pause()*, *sigsuspend()*, *sleep()*,  
 4225 or *read()/write()* on a slow device like a terminal) that are interruptible. This permits  
 4226 applications to respond to interactive signals or to set timeouts on calls to most such functions  
 4227 with *alarm()*. Therefore, implementations should generally make such functions (including ones  
 4228 defined as extensions) interruptible.

4229 Functions not mentioned explicitly as interruptible may be so on some implementations,  
 4230 possibly as an extension where the function gives an [EINTR] error. There are several functions  
 4231 (for example, *getpid()*, *getuid()*) that are specified as never returning an error, which can thus  
 4232 never be extended in this way.

4233 If a signal-catching function returns while the SA\_RESTART flag is in effect, an interrupted  
 4234 function is restarted at the point it was interrupted. Conforming applications cannot make  
 4235 assumptions about the internal behavior of interrupted functions, even if the functions are  
 4236 async-signal-safe. For example, suppose the *read()* function is interrupted with SA\_RESTART in  
 4237 effect, the signal-catching function closes the file descriptor being read from and returns, and the  
 4238 *read()* function is then restarted; in this case the application cannot assume that the *read()*  
 4239 function will give an [EBADF] error, since *read()* might have checked the file descriptor for  
 4240 validity before being interrupted.

## 4241 **B.2.5 Standard I/O Streams**

### 4242 *B.2.5.1 Interaction of File Descriptors and Standard I/O Streams*

4243 There is no additional rationale provided for this section.

### 4244 *B.2.5.2 Stream Orientation and Encoding Rules*

4245 There is no additional rationale provided for this section.

## 4246 **B.2.6 STREAMS**

4247 STREAMS are introduced into IEEE Std 1003.1-2001 as part of the alignment with the Single  
 4248 UNIX Specification, but marked as an option in recognition that not all systems may wish to  
 4249 implement the facility. The option within IEEE Std 1003.1-2001 is denoted by the XSR margin  
 4250 marker. The standard developers made this option independent of the XSI option.

4251 STREAMS are a method of implementing network services and other character-based  
 4252 input/output mechanisms, with the STREAM being a full-duplex connection between a process  
 4253 and a device. STREAMS provides direct access to protocol modules, and optional protocol  
 4254 modules can be interposed between the process-end of the STREAM and the device-driver at the  
 4255 device-end of the STREAM. Pipes can be implemented using the STREAMS mechanism, so they  
 4256 can provide process-to-process as well as process-to-device communications.

4257 This section introduces STREAMS I/O, the message types used to control them, an overview of  
 4258 the priority mechanism, and the interfaces used to access them.

### 4259 *B.2.6.1 Accessing STREAMS*

4260 There is no additional rationale provided for this section.

## 4261 **B.2.7 XSI Interprocess Communication**

4262 There are two forms of IPC supported as options in IEEE Std 1003.1-2001. The traditional  
 4263 System V IPC routines derived from the SVID—that is, the *msg\*()*, *sem\*()*, and *shm\*()*  
 4264 interfaces—are mandatory on XSI-conformant systems. Thus, all XSI-conformant systems  
 4265 provide the same mechanisms for manipulating messages, shared memory, and semaphores.

4266 In addition, the POSIX Realtime Extension provides an alternate set of routines for those systems  
 4267 supporting the appropriate options.

4268 The application writer is presented with a choice: the System V interfaces or the POSIX  
 4269 interfaces (loosely derived from the Berkeley interfaces). The XSI profile prefers the System V

4270 interfaces, but the POSIX interfaces may be more suitable for realtime or other performance-  
4271 sensitive applications.

#### 4272 **B.2.7.1 IPC General Information**

4273 General information that is shared by all three mechanisms is described in this section. The  
4274 common permissions mechanism is briefly introduced, describing the mode bits, and how they  
4275 are used to determine whether or not a process has access to read or write/alter the appropriate  
4276 instance of one of the IPC mechanisms. All other relevant information is contained in the  
4277 reference pages themselves.

4278 The semaphore type of IPC allows processes to communicate through the exchange of  
4279 semaphore values. A semaphore is a positive integer. Since many applications require the use of  
4280 more than one semaphore, XSI-conformant systems have the ability to create sets or arrays of  
4281 semaphores.

4282 Calls to support semaphores include:

4283 *semctl()*, *semget()*, *semop()*

4284 Semaphore sets are created by using the *semget()* function.

4285 The message type of IPC allows processes to communicate through the exchange of data stored  
4286 in buffers. This data is transmitted between processes in discrete portions known as messages.

4287 Calls to support message queues include:

4288 *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*

4289 The shared memory type of IPC allows two or more processes to share memory and  
4290 consequently the data contained therein. This is done by allowing processes to set up access to a  
4291 common memory address space. This sharing of memory provides a fast means of exchange of  
4292 data between processes.

4293 Calls to support shared memory include:

4294 *shmctl()*, *shmdt()*, *shmget()*

4295 The *ftok()* interface is also provided.

#### 4296 **B.2.8 Realtime**

##### 4297 **Advisory Information**

4298 POSIX.1b contains an Informative Annex with proposed interfaces for “realtime files”. These  
4299 interfaces could determine groups of the exact parameters required to do “direct I/O” or  
4300 “extents”. These interfaces were objected to by a significant portion of the balloting group as too  
4301 complex. A conforming application had little chance of correctly navigating the large parameter  
4302 space to match its desires to the system. In addition, they only applied to a new type of file  
4303 (realtime files) and they told the implementation exactly what to do as opposed to advising the  
4304 implementation on application behavior and letting it optimize for the system the (portable)  
4305 application was running on. For example, it was not clear how a system that had a disk array  
4306 should set its parameters.

4307 There seemed to be several overall goals:

- 4308 • Optimizing sequential access
- 4309 • Optimizing caching behavior

- 4310 • Optimizing I/O data transfer
- 4311 • Preallocation

4312 The advisory interfaces, *posix\_fadvise()* and *posix\_madvise()*, satisfy the first two goals. The  
 4313 POSIX\_FADV\_SEQUENTIAL and POSIX\_MADV\_SEQUENTIAL advice tells the  
 4314 implementation to expect serial access. Typically the system will prefetch the next several serial  
 4315 accesses in order to overlap I/O. It may also free previously accessed serial data if memory is  
 4316 tight. If the application is not doing serial access it can use POSIX\_FADV\_WILLNEED and  
 4317 POSIX\_MADV\_WILLNEED to accomplish I/O overlap, as required. When the application  
 4318 advises POSIX\_FADV\_RANDOM or POSIX\_MADV\_RANDOM behavior, the implementation  
 4319 usually tries to fetch a minimum amount of data with each request and it does not expect much  
 4320 locality. POSIX\_FADV\_DONTNEED and POSIX\_MADV\_DONTNEED allow the system to free  
 4321 up caching resources as the data will not be required in the near future.

4322 POSIX\_FADV\_NOREUSE tells the system that caching the specified data is not optimal. For file  
 4323 I/O, the transfer should go directly to the user buffer instead of being cached internally by the  
 4324 implementation. To portably perform direct disk I/O on all systems, the application must  
 4325 perform its I/O transfers according to the following rules:

- 4326 1. The user buffer should be aligned according to the {POSIX\_REC\_XFER\_ALIGN} *pathconf()*  
 4327 variable.
- 4328 2. The number of bytes transferred in an I/O operation should be a multiple of the  
 4329 {POSIX\_ALLOC\_SIZE\_MIN} *pathconf()* variable.
- 4330 3. The offset into the file at the start of an I/O operation should be a multiple of the  
 4331 {POSIX\_ALLOC\_SIZE\_MIN} *pathconf()* variable.
- 4332 4. The application should ensure that all threads which open a given file specify  
 4333 POSIX\_FADV\_NOREUSE to be sure that there is no unexpected interaction between  
 4334 threads using buffered I/O and threads using direct I/O to the same file.

4335 In some cases, a user buffer must be properly aligned in order to be transferred directly to/from  
 4336 the device. The {POSIX\_REC\_XFER\_ALIGN} *pathconf()* variable tells the application the proper  
 4337 alignment.

4338 The preallocation goal is met by the space control function, *posix\_fallocate()*. The application can  
 4339 use *posix\_fallocate()* to guarantee no [ENOSPC] errors and to improve performance by prepaying  
 4340 any overhead required for block allocation.

4341 Implementations may use information conveyed by a previous *posix\_fadvise()* call to influence  
 4342 the manner in which allocation is performed. For example, if an application did the following  
 4343 calls:

```
4344     fd = open("file");
4345     posix_fadvise(fd, offset, len, POSIX_FADV_SEQUENTIAL);
4346     posix_fallocate(fd, len, size);
```

4347 an implementation might allocate the file contiguously on disk.

4348 Finally, the *pathconf()* variables {POSIX\_REC\_MIN\_XFER\_SIZE},  
 4349 {POSIX\_REC\_MAX\_XFER\_SIZE}, and {POSIX\_REC\_INCR\_XFER\_SIZE} tell the application a  
 4350 range of transfer sizes that are recommended for best I/O performance.

4351 Where bounded response time is required, the vendor can supply the appropriate settings of the  
 4352 advisories to achieve a guaranteed performance level.

4353 The interfaces meet the goals while allowing applications using regular files to take advantage of  
 4354 performance optimizations. The interfaces tell the implementation expected application

behavior which the implementation can use to optimize performance on a particular system with a particular dynamic load.

The *posix\_memalign()* function was added to allow for the allocation of specifically aligned buffers; for example, for {POSIX\_REC\_XFER\_ALIGN}.

The working group also considered the alternative of adding a function which would return an aligned pointer to memory within a user-supplied buffer. This was not considered to be the best method, because it potentially wastes large amounts of memory when buffers need to be aligned on large alignment boundaries.

## Message Passing

This section provides the rationale for the definition of the message passing interface in IEEE Std 1003.1-2001. This is presented in terms of the objectives, models, and requirements imposed upon this interface.

### • Objectives

Many applications, including both realtime and database applications, require a means of passing arbitrary amounts of data between cooperating processes comprising the overall application on one or more processors. Many conventional interfaces for interprocess communication are insufficient for realtime applications in that efficient and deterministic data passing methods cannot be implemented. This has prompted the definition of message passing interfaces providing these facilities:

- Open a message queue.
- Send a message to a message queue.
- Receive a message from a queue, either synchronously or asynchronously.
- Alter message queue attributes for flow and resource control.

It is assumed that an application may consist of multiple cooperating processes and that these processes may wish to communicate and coordinate their activities. The message passing facility described in IEEE Std 1003.1-2001 allows processes to communicate through system-wide queues. These message queues are accessed through names that may be pathnames. A message queue can be opened for use by multiple sending and/or multiple receiving processes.

### • Background on Embedded Applications

Interprocess communication utilizing message passing is a key facility for the construction of deterministic, high-performance realtime applications. The facility is present in all realtime systems and is the framework upon which the application is constructed. The performance of the facility is usually a direct indication of the performance of the resulting application.

Realtime applications, especially for embedded systems, are typically designed around the performance constraints imposed by the message passing mechanisms. Applications for embedded systems are typically very tightly constrained. Application writers expect to design and control the entire system. In order to minimize system costs, the writer will attempt to use all resources to their utmost and minimize the requirement to add additional memory or processors.

The embedded applications usually share address spaces and only a simple message passing mechanism is required. The application can readily access common data incurring only mutual-exclusion overheads. The models desired are the simplest possible with the application building higher-level facilities only when needed.



- Requirements

The following requirements determined the features of the message passing facilities defined in IEEE Std 1003.1-2001:

- Naming of Message Queues

The mechanism for gaining access to a message queue is a pathname evaluated in a context that is allowed to be a file system name space, or it can be independent of any file system. This is a specific attempt to allow implementations based on either method in order to address both embedded systems and to also allow implementation in larger systems.

The interface of *mq\_open()* is defined to allow but not require the access control and name conflicts resulting from utilizing a file system for name resolution. All required behavior is specified for the access control case. Yet a conforming implementation, such as an embedded system kernel, may define that there are no distinctions between users and may define that all processes have all access privileges.

- Embedded System Naming

Embedded systems need to be able to utilize independent name spaces for accessing the various system objects. They typically do not have a file system, precluding its utilization as a common name resolution mechanism. The modularity of an embedded system limits the connections between separate mechanisms that can be allowed.

Embedded systems typically do not have any access protection. Since the system does not support the mixing of applications from different areas, and usually does not even have the concept of an authorization entity, access control is not useful.

- Large System Naming

On systems with more functionality, the name resolution must support the ability to use the file system as the name resolution mechanism/object storage medium and to have control over access to the objects. Utilizing the pathname space can result in further errors when the names conflict with other objects.

- Fixed Size of Messages

The interfaces impose a fixed upper bound on the size of messages that can be sent to a specific message queue. The size is set on an individual queue basis and cannot be changed dynamically.

The purpose of the fixed size is to increase the ability of the system to optimize the implementation of *mq\_send()* and *mq\_receive()*. With fixed sizes of messages and fixed numbers of messages, specific message blocks can be pre-allocated. This eliminates a significant amount of checking for errors and boundary conditions. Additionally, an implementation can optimize data copying to maximize performance. Finally, with a restricted range of message sizes, an implementation is better able to provide deterministic operations.

- Prioritization of Messages

Message prioritization allows the application to determine the order in which messages are received. Prioritization of messages is a key facility that is provided by most realtime kernels and is heavily utilized by the applications. The major purpose of having priorities in message queues is to avoid priority inversions in the message system, where a high-priority message is delayed behind one or more lower-priority messages. This allows the applications to be designed so that they do not need to be interrupted in order to change

the flow of control when exceptional conditions occur. The prioritization does add additional overhead to the message operations in those cases it is actually used but a clever implementation can optimize for the FIFO case to make that more efficient.

#### — Asynchronous Notification

The interface supports the ability to have a task asynchronously notified of the availability of a message on the queue. The purpose of this facility is to allow the task to perform other functions and yet still be notified that a message has become available on the queue.

To understand the requirement for this function, it is useful to understand two models of application design: a single task performing multiple functions and multiple tasks performing a single function. Each of these models has advantages.

Asynchronous notification is required to build the model of a single task performing multiple operations. This model typically results from either the expectation that interruption is less expensive than utilizing a separate task or from the growth of the application to include additional functions.

### Semaphores

Semaphores are a high-performance process synchronization mechanism. Semaphores are named by null-terminated strings of characters.

A semaphore is created using the *sem\_init()* function or the *sem\_open()* function with the *O\_CREAT* flag set in *oflag*.

To use a semaphore, a process has to first initialize the semaphore or inherit an open descriptor for the semaphore via *fork()*.

A semaphore preserves its state when the last reference is closed. For example, if a semaphore has a value of 13 when the last reference is closed, it will have a value of 13 when it is next opened.

When a semaphore is created, an initial state for the semaphore has to be provided. This value is a non-negative integer. Negative values are not possible since they indicate the presence of blocked processes. The persistence of any of these objects across a system crash or a system reboot is undefined. Conforming applications must not depend on any sort of persistence across a system reboot or a system crash.

#### • Models and Requirements

A realtime system requires synchronization and communication between the processes comprising the overall application. An efficient and reliable synchronization mechanism has to be provided in a realtime system that will allow more than one schedulable process mutually-exclusive access to the same resource. This synchronization mechanism has to allow for the optimal implementation of synchronization or systems implementors will define other, more cost-effective methods.

At issue are the methods whereby multiple processes (tasks) can be designed and implemented to work together in order to perform a single function. This requires interprocess communication and synchronization. A semaphore mechanism is the lowest level of synchronization that can be provided by an operating system.

A semaphore is defined as an object that has an integral value and a set of blocked processes associated with it. If the value is positive or zero, then the set of blocked processes is empty; otherwise, the size of the set is equal to the absolute value of the semaphore value. The value of the semaphore can be incremented or decremented by any process with access to the

semaphore and must be done as an indivisible operation. When a semaphore value is less than or equal to zero, any process that attempts to lock it again will block or be informed that it is not possible to perform the operation.

A semaphore may be used to guard access to any resource accessible by more than one schedulable task in the system. It is a global entity and not associated with any particular process. As such, a method of obtaining access to the semaphore has to be provided by the operating system. A process that wants access to a critical resource (section) has to wait on the semaphore that guards that resource. When the semaphore is locked on behalf of a process, it knows that it can utilize the resource without interference by any other cooperating process in the system. When the process finishes its operation on the resource, leaving it in a well-defined state, it posts the semaphore, indicating that some other process may now obtain the resource associated with that semaphore.

In this section, mutexes and condition variables are specified as the synchronization mechanisms between threads.

These primitives are typically used for synchronizing threads that share memory in a single process. However, this section provides an option allowing the use of these synchronization interfaces and objects between processes that share memory, regardless of the method for sharing memory.

Much experience with semaphores shows that there are two distinct uses of synchronization: locking, which is typically of short duration; and waiting, which is typically of long or unbounded duration. These distinct usages map directly onto mutexes and condition variables, respectively.

Semaphores are provided in IEEE Std 1003.1-2001 primarily to provide a means of synchronization for processes; these processes may or may not share memory. Mutexes and condition variables are specified as synchronization mechanisms between threads; these threads always share (some) memory. Both are synchronization paradigms that have been in widespread use for a number of years. Each set of primitives is particularly well matched to certain problems.

With respect to binary semaphores, experience has shown that condition variables and mutexes are easier to use for many synchronization problems than binary semaphores. The primary reason for this is the explicit appearance of a Boolean predicate that specifies when the condition wait is satisfied. This Boolean predicate terminates a loop, including the call to *pthread\_cond\_wait()*. As a result, extra wakeups are benign since the predicate governs whether the thread will actually proceed past the condition wait. With stateful primitives, such as binary semaphores, the wakeup in itself typically means that the wait is satisfied. The burden of ensuring correctness for such waits is thus placed on *all* signalers of the semaphore rather than on an *explicitly coded* Boolean predicate located at the condition wait. Experience has shown that the latter creates a major improvement in safety and ease-of-use.

Counting semaphores are well matched to dealing with producer/consumer problems, including those that might exist between threads of different processes, or between a signal handler and a thread. In the former case, there may be little or no memory shared by the processes; in the latter case, one is not communicating between co-equal threads, but between a thread and an interrupt-like entity. It is for these reasons that IEEE Std 1003.1-2001 allows semaphores to be used by threads.

Mutexes and condition variables have been effectively used with and without priority inheritance, priority ceiling, and other attributes to synchronize threads that share memory. The efficiency of their implementation is comparable to or better than that of other synchronization primitives that are sometimes harder to use (for example, binary

semaphores). Furthermore, there is at least one known implementation of Ada tasking that uses these primitives. Mutexes and condition variables together constitute an appropriate, sufficient, and complete set of inter-thread synchronization primitives.

Efficient multi-threaded applications require high-performance synchronization primitives. Considerations of efficiency and generality require a small set of primitives upon which more sophisticated synchronization functions can be built.

- Standardization Issues

It is possible to implement very high-performance semaphores using test-and-set instructions on shared memory locations. The library routines that implement such a high-performance interface have to properly ensure that a *sem\_wait()* or *sem\_trywait()* operation that cannot be performed will issue a blocking semaphore system call or properly report the condition to the application. The same interface to the application program would be provided by a high-performance implementation.

### B.2.8.1 Realtime Signals

#### Realtime Signals Extension

This portion of the rationale presents models, requirements, and standardization issues relevant to the Realtime Signals Extension. This extension provides the capability required to support reliable, deterministic, asynchronous notification of events. While a new mechanism, unencumbered by the historical usage and semantics of POSIX.1 signals, might allow for a more efficient implementation, the application requirements for event notification can be met with a small number of extensions to signals. Therefore, a minimal set of extensions to signals to support the application requirements is specified.

The realtime signal extensions specified in this section are used by other realtime functions requiring asynchronous notification:

- Models

The model supported is one of multiple cooperating processes, each of which handles multiple asynchronous external events. Events represent occurrences that are generated as the result of some activity in the system. Examples of occurrences that can constitute an event include:

- Completion of an asynchronous I/O request
- Expiration of a POSIX.1b timer
- Arrival of an interprocess message
- Generation of a user-defined event

Processing of these events may occur synchronously via polling for event notifications or asynchronously via a software interrupt mechanism. Existing practice for this model is well established for traditional proprietary realtime operating systems, realtime executives, and realtime extended POSIX-like systems.

A contrasting model is that of “cooperating sequential processes” where each process handles a single priority of events via polling. Each process blocks while waiting for events, and each process depends on the preemptive, priority-based process scheduling mechanism to arbitrate between events of different priority that need to be processed concurrently. Existing practice for this model is also well established for small realtime executives that typically execute in an unprotected physical address space, but it is just emerging in the

context of a fuller function operating system with multiple virtual address spaces.

It could be argued that the cooperating sequential process model, and the facilities supported by the POSIX Threads Extension obviate a software interrupt model. But, even with the cooperating sequential process model, the need has been recognized for a software interrupt model to handle exceptional conditions and process aborting, so the mechanism must be supported in any case. Furthermore, it is not the purview of IEEE Std 1003.1-2001 to attempt to convince realtime practitioners that their current application models based on software interrupts are “broken” and should be replaced by the cooperating sequential process model. Rather, it is the charter of IEEE Std 1003.1-2001 to provide standard extensions to mechanisms that support existing realtime practice.

- Requirements

This section discusses the following realtime application requirements for asynchronous event notification:

- Reliable delivery of asynchronous event notification

The events notification mechanism guarantees delivery of an event notification. Asynchronous operations (such as asynchronous I/O and timers) that complete significantly after they are invoked have to guarantee that delivery of the event notification can occur at the time of completion.

- Prioritized handling of asynchronous event notifications

The events notification mechanism supports the assigning of a user function as an event notification handler. Furthermore, the mechanism supports the preemption of an event handler function by a higher priority event notification and supports the selection of the highest priority pending event notification when multiple notifications (of different priority) are pending simultaneously.

The model here is based on hardware interrupts. Asynchronous event handling allows the application to ensure that time-critical events are immediately processed when delivered, without the indeterminism of being at a random location within a polling loop. Use of handler priority allows the specification of how handlers are interrupted by other higher priority handlers.

- Differentiation between multiple occurrences of event notifications of the same type

The events notification mechanism passes an application-defined value to the event handler function. This value can be used for a variety of purposes, such as enabling the application to identify which of several possible events of the same type (for example, timer expirations) has occurred.

- Polled reception of asynchronous event notifications

The events notification mechanism supports blocking and non-blocking polls for asynchronous event notification.

The polled mode of operation is often preferred over the interrupt mode by those practitioners accustomed to this model. Providing support for this model facilitates the porting of applications based on this model to POSIX.1b conforming systems.

- Deterministic response to asynchronous event notifications

The events notification mechanism does not preclude implementations that provide deterministic event dispatch latency and minimizes the number of system calls needed to use the event facilities during realtime processing.

- Rationale for Extension

POSIX.1 signals have many of the characteristics necessary to support the asynchronous handling of event notifications, and the Realtime Signals Extension addresses the following deficiencies in the POSIX.1 signal mechanism:

- Signals do not support reliable delivery of event notification. Subsequent occurrences of a pending signal are not guaranteed to be delivered.
- Signals do not support prioritized delivery of event notifications. The order of signal delivery when multiple unblocked signals are pending is undefined.
- Signals do not support the differentiation between multiple signals of the same type.

### B.2.8.2 Asynchronous I/O

Many applications need to interact with the I/O subsystem in an asynchronous manner. The asynchronous I/O mechanism provides the ability to overlap application processing and I/O operations initiated by the application. The asynchronous I/O mechanism allows a single process to perform I/O simultaneously to a single file multiple times or to multiple files multiple times.

#### Overview

Asynchronous I/O operations proceed in logical parallel with the processing done by the application after the asynchronous I/O has been initiated. Other than this difference, asynchronous I/O behaves similarly to normal I/O using *read()*, *write()*, *lseek()*, and *fsync()*. The effect of issuing an asynchronous I/O request is as if a separate thread of execution were to perform atomically the implied *lseek()* operation, if any, and then the requested I/O operation (either *read()*, *write()*, or *fsync()*). There is no seek implied with a call to *aio\_fsync()*. Concurrent asynchronous operations and synchronous operations applied to the same file update the file as if the I/O operations had proceeded serially.

When asynchronous I/O completes, a signal can be delivered to the application to indicate the completion of the I/O. This signal can be used to indicate that buffers and control blocks used for asynchronous I/O can be reused. Signal delivery is not required for an asynchronous operation and may be turned off on a per-operation basis by the application. Signals may also be synchronously polled using *aio\_suspend()*, *sigtimedwait()*, or *sigwaitinfo()*.

Normal I/O has a return value and an error status associated with it. Asynchronous I/O returns a value and an error status when the operation is first submitted, but that only relates to whether the operation was successfully queued up for servicing. The I/O operation itself also has a return status and an error value. To allow the application to retrieve the return status and the error value, functions are provided that, given the address of an asynchronous I/O control block, yield the return and error status associated with the operation. Until an asynchronous I/O operation is done, its error status is [EINPROGRESS]. Thus, an application can poll for completion of an asynchronous I/O operation by waiting for the error status to become equal to a value other than [EINPROGRESS]. The return status of an asynchronous I/O operation is undefined so long as the error status is equal to [EINPROGRESS].

Storage for asynchronous operation return and error status may be limited. Submission of asynchronous I/O operations may fail if this storage is exceeded. When an application retrieves the return status of a given asynchronous operation, therefore, any system-maintained storage used for this status and the error status may be reclaimed for use by other asynchronous operations.

Asynchronous I/O can be performed on file descriptors that have been enabled for POSIX.1b synchronized I/O. In this case, the I/O operation still occurs asynchronously, as defined herein; however, the asynchronous operation I/O in this case is not completed until the I/O has reached either the state of synchronized I/O data integrity completion or synchronized I/O file integrity completion, depending on the sort of synchronized I/O that is enabled on the file descriptor.

## Models

Three models illustrate the use of asynchronous I/O: a journalization model, a data acquisition model, and a model of the use of asynchronous I/O in supercomputing applications.

- Journalization Model

Many realtime applications perform low-priority journalizing functions. Journalizing requires that logging records be queued for output without blocking the initiating process.

- Data Acquisition Model

A data acquisition process may also serve as a model. The process has two or more channels delivering intermittent data that must be read within a certain time. The process issues one asynchronous read on each channel. When one of the channels needs data collection, the process reads the data and posts it through an asynchronous write to secondary memory for future processing.

- Supercomputing Model

The supercomputing community has used asynchronous I/O much like that specified in POSIX.1 for many years. This community requires the ability to perform multiple I/O operations to multiple devices with a minimal number of entries to “the system”; each entry to “the system” provokes a major delay in operations when compared to the normal progress made by the application. This existing practice motivated the use of combined *lseek()* and *read()* or *write()* calls, as well as the *lio\_listio()* call. Another common practice is to disable signal notification for I/O completion, and simply poll for I/O completion at some interval by which the I/O should be completed. Likewise, interfaces like *aio\_cancel()* have been in successful commercial use for many years. Note also that an underlying implementation of asynchronous I/O will require the ability, at least internally, to cancel outstanding asynchronous I/O, at least when the process exits. (Consider an asynchronous read from a terminal, when the process intends to exit immediately.)

## Requirements

Asynchronous input and output for realtime implementations have these requirements:

- The ability to queue multiple asynchronous read and write operations to a single open instance. Both sequential and random access should be supported.
- The ability to queue asynchronous read and write operations to multiple open instances.
- The ability to obtain completion status information by polling and/or asynchronous event notification.
- Asynchronous event notification on asynchronous I/O completion is optional.
- It has to be possible for the application to associate the event with the *aiochp* for the operation that generated the event.
- The ability to cancel queued requests.
- The ability to wait upon asynchronous I/O completion in conjunction with other types of events.

- The ability to accept an *aio\_read()* and an *aio\_cancel()* for a device that accepts a *read()*, and the ability to accept an *aio\_write()* and an *aio\_cancel()* for a device that accepts a *write()*. This does not imply that the operation is asynchronous.

## Standardization Issues

The following issues are addressed by the standardization of asynchronous I/O:

- Rationale for New Interface

Non-blocking I/O does not satisfy the needs of either realtime or high-performance computing models; these models require that a process overlap program execution and I/O processing. Realtime applications will often make use of direct I/O to or from the address space of the process, or require synchronized (unbuffered) I/O; they also require the ability to overlap this I/O with other computation. In addition, asynchronous I/O allows an application to keep a device busy at all times, possibly achieving greater throughput. Supercomputing and database architectures will often have specialized hardware that can provide true asynchrony underlying the logical asynchrony provided by this interface. In addition, asynchronous I/O should be supported by all types of files and devices in the same manner.

- Effect of Buffering

If asynchronous I/O is performed on a file that is buffered prior to being actually written to the device, it is possible that asynchronous I/O will offer no performance advantage over normal I/O; the cycles *stolen* to perform the asynchronous I/O will be taken away from the running process and the I/O will occur at interrupt time. This potential lack of gain in performance in no way obviates the need for asynchronous I/O by realtime applications, which very often will use specialized hardware support, multiple processors, and/or unbuffered, synchronized I/O.

### B.2.8.3 Memory Management

All memory management and shared memory definitions are located in the `<sys/mman.h>` header. This is for alignment with historical practice.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/7 is applied, correcting the shading and margin markers in the introduction to Section 2.8.3.1.

## Memory Locking Functions

This portion of the rationale presents models, requirements, and standardization issues relevant to process memory locking.

- Models

Realtime systems that conform to IEEE Std 1003.1-2001 are expected (and desired) to be supported on systems with demand-paged virtual memory management, non-paged swapping memory management, and physical memory systems with no memory management hardware. The general case, however, is the demand-paged, virtual memory system with each POSIX process running in a virtual address space. Note that this includes architectures where each process resides in its own virtual address space and architectures where the address space of each process is only a portion of a larger global virtual address space.

The concept of memory locking is introduced to eliminate the indeterminacy introduced by paging and swapping, and to support an upper bound on the time required to access the memory mapped into the address space of a process. Ideally, this upper bound will be the



same as the time required for the processor to access “main memory”, including any address translation and cache miss overheads. But some implementations—primarily on mainframes—will not actually force locked pages to be loaded and held resident in main memory. Rather, they will handle locked pages so that accesses to these pages will meet the performance metrics for locked process memory in the implementation. Also, although it is not, for example, the intention that this interface, as specified, be used to lock process memory into “cache”, it is conceivable that an implementation could support a large static RAM memory and define this as “main memory” and use a large[r] dynamic RAM as “backing store”. These interfaces could then be interpreted as supporting the locking of process memory into the static RAM. Support for multiple levels of backing store would require extensions to these interfaces.

Implementations may also use memory locking to guarantee a fixed translation between virtual and physical addresses where such is beneficial to improving determinancy for direct-to/from-process input/output. IEEE Std 1003.1-2001 does not guarantee to the application that the virtual-to-physical address translations, if such exist, are fixed, because such behavior would not be implementable on all architectures on which implementations of IEEE Std 1003.1-2001 are expected. But IEEE Std 1003.1-2001 does mandate that an implementation define, for the benefit of potential users, whether or not locking guarantees fixed translations.

Memory locking is defined with respect to the address space of a process. Only the pages mapped into the address space of a process may be locked by the process, and when the pages are no longer mapped into the address space—for whatever reason—the locks established with respect to that address space are removed. Shared memory areas warrant special mention, as they may be mapped into more than one address space or mapped more than once into the address space of a process; locks may be established on pages within these areas with respect to several of these mappings. In such a case, the lock state of the underlying physical pages is the logical OR of the lock state with respect to each of the mappings. Only when all such locks have been removed are the shared pages considered unlocked.

In recognition of the page granularity of Memory Management Units (MMU), and in order to support locking of ranges of address space, memory locking is defined in terms of “page” granularity. That is, for the interfaces that support an address and size specification for the region to be locked, the address must be on a page boundary, and all pages mapped by the specified range are locked, if valid. This means that the length is implicitly rounded up to a multiple of the page size. The page size is implementation-defined and is available to applications as a compile-time symbolic constant or at runtime via *sysconf()*.

A “real memory” POSIX.1b implementation that has no MMU could elect not to support these interfaces, returning [ENOSYS]. But an application could easily interpret this as meaning that the implementation would unconditionally page or swap the application when such is not the case. It is the intention of IEEE Std 1003.1-2001 that such a system could define these interfaces as “NO-OPs”, returning success without actually performing any function except for mandated argument checking.

- Requirements

For realtime applications, memory locking is generally considered to be required as part of application initialization. This locking is performed after an application has been loaded (that is, *exec'd*) and the program remains locked for its entire lifetime. But to support applications that undergo major mode changes where, in one mode, locking is required, but in another it is not, the specified interfaces allow repeated locking and unlocking of memory within the lifetime of a process.

When a realtime application locks its address space, it should not be necessary for the application to then “touch” all of the pages in the address space to guarantee that they are resident or else suffer potential paging delays the first time the page is referenced. Thus, IEEE Std 1003.1-2001 requires that the pages locked by the specified interfaces be resident when the locking functions return successfully.

Many architectures support system-managed stacks that grow automatically when the current extent of the stack is exceeded. A realtime application has a requirement to be able to “preallocate” sufficient stack space and lock it down so that it will not suffer page faults to grow the stack during critical realtime operation. There was no consensus on a portable way to specify how much stack space is needed, so IEEE Std 1003.1-2001 supports no specific interface for preallocating stack space. But an application can portably lock down a specific amount of stack space by specifying `MCL_FUTURE` in a call to `mlockall()` and then calling a dummy function that declares an automatic array of the desired size.

Memory locking for realtime applications is also generally considered to be an “all or nothing” proposition. That is, the entire process, or none, is locked down. But, for applications that have well-defined sections that need to be locked and others that do not, IEEE Std 1003.1-2001 supports an optional set of interfaces to lock or unlock a range of process addresses. Reasons for locking down a specific range include:

- An asynchronous event handler function that must respond to external events in a deterministic manner such that page faults cannot be tolerated
- An input/output “buffer” area that is the target for direct-to-process I/O, and the overhead of implicit locking and unlocking for each I/O call cannot be tolerated

Finally, locking is generally viewed as an “application-wide” function. That is, the application is globally aware of which regions are locked and which are not over time. This is in contrast to a function that is used temporarily within a “third party” library routine whose function is unknown to the application, and therefore must have no “side effects”. The specified interfaces, therefore, do not support “lock stacking” or “lock nesting” within a process. But, for pages that are shared between processes or mapped more than once into a process address space, “lock stacking” is essentially mandated by the requirement that unlocking of pages that are mapped by more than one process or more than once by the same process does not affect locks established on the other mappings.

There was some support for “lock stacking” so that locking could be transparently used in functions or opaque modules. But the consensus was not to burden all implementations with lock stacking (and reference counting), and an implementation option was proposed. There were strong objections to the option because applications would have to support both options in order to remain portable. The consensus was to eliminate lock stacking altogether, primarily through overwhelming support for the System V “`m[un]lock[all]`” interface on which IEEE Std 1003.1-2001 is now based.

Locks are not inherited across `fork()`s because some implementations implement `fork()` by creating new address spaces for the child. In such an implementation, requiring locks to be inherited would lead to new situations in which a fork would fail due to the inability of the system to lock sufficient memory to lock both the parent and the child. The consensus was that there was no benefit to such inheritance. Note that this does not mean that locks are removed when, for instance, a thread is created in the same address space.

Similarly, locks are not inherited across `exec` because some implementations implement `exec` by unmapping all of the pages in the address space (which, by definition, removes the locks on these pages), and maps in pages of the `exec'd` image. In such an implementation, requiring locks to be inherited would lead to new situations in which `exec` would fail. Reporting this

failure would be very cumbersome to detect in time to report to the calling process, and no appropriate mechanism exists for informing the *exec'd* process of its status.

It was determined that, if the newly loaded application required locking, it was the responsibility of that application to establish the locks. This is also in keeping with the general view that it is the responsibility of the application to be aware of all locks that are established.

There was one request to allow (not mandate) locks to be inherited across *fork()*, and a request for a flag, *MCL\_INHERIT*, that would specify inheritance of memory locks across *execs*. Given the difficulties raised by this and the general lack of support for the feature in IEEE Std 1003.1-2001, it was not added. IEEE Std 1003.1-2001 does not preclude an implementation from providing this feature for administrative purposes, such as a “run” command that will lock down and execute a specified application. Additionally, the rationale for the objection equated *fork()* with creating a thread in the address space. IEEE Std 1003.1-2001 does not mandate releasing locks when creating additional threads in an existing process.

- **Standardization Issues**

One goal of IEEE Std 1003.1-2001 is to define a set of primitives that provide the necessary functionality for realtime applications, with consideration for the needs of other application domains where such were identified, which is based to the extent possible on existing industry practice.

The Memory Locking option is required by many realtime applications to tune performance. Such a facility is accomplished by placing constraints on the virtual memory system to limit paging of time of the process or of critical sections of the process. This facility should not be used by most non-realtime applications.

Optional features provided in IEEE Std 1003.1-2001 allow applications to lock selected address ranges with the caveat that the process is responsible for being aware of the page granularity of locking and the unnested nature of the locks.

## **Mapped Files Functions**

The Memory Mapped Files option provides a mechanism that allows a process to access files by directly incorporating file data into its address space. Once a file is “mapped” into a process address space, the data can be manipulated by instructions as memory. The use of mapped files can significantly reduce I/O data movement since file data does not have to be copied into process data buffers as in *read()* and *write()*. If more than one process maps a file, its contents are shared among them. This provides a low overhead mechanism by which processes can synchronize and communicate.

- **Historical Perspective**

Realtime applications have historically been implemented using a collection of cooperating processes or tasks. In early systems, these processes ran on bare hardware (that is, without an operating system) with no memory relocation or protection. The application paradigms that arose from this environment involve the sharing of data between the processes.

When realtime systems were implemented on top of vendor-supplied operating systems, the paradigm or performance benefits of direct access to data by multiple processes was still deemed necessary. As a result, operating systems that claim to support realtime applications must support the shared memory paradigm.

Additionally, a number of realtime systems provide the ability to map specific sections of the physical address space into the address space of a process. This ability is required if an

application is to obtain direct access to memory locations that have specific properties (for example, refresh buffers or display devices, dual ported memory locations, DMA target locations). The use of this ability is common enough to warrant some degree of standardization of its interface. This ability overlaps the general paradigm of shared memory in that, in both instances, common global objects are made addressable by individual processes or tasks.

Finally, a number of systems also provide the ability to map process addresses to files. This provides both a general means of sharing persistent objects, and using files in a manner that optimizes memory and swapping space usage.

Simple shared memory is clearly a special case of the more general file mapping capability. In addition, there is relatively widespread agreement and implementation of the file mapping interface. In these systems, many different types of objects can be mapped (for example, files, memory, devices, and so on) using the same mapping interfaces. This approach both minimizes interface proliferation and maximizes the generality of programs using the mapping interfaces.

- Memory Mapped Files Usage

A memory object can be concurrently mapped into the address space of one or more processes. The *mmap()* and *munmap()* functions allow a process to manipulate their address space by mapping portions of memory objects into it and removing them from it. When multiple processes map the same memory object, they can share access to the underlying data. Implementations may restrict the size and alignment of mappings to be on *page-size* boundaries. The page size, in bytes, is the value of the system-configurable variable {PAGESIZE}, typically accessed by calling *sysconf()* with a *name* argument of *\_SC\_PAGESIZE*. If an implementation has no restrictions on size or alignment, it may specify a 1-byte page size.

To map memory, a process first opens a memory object. The *ftruncate()* function can be used to contract or extend the size of the memory object even when the object is currently mapped. If the memory object is extended, the contents of the extended areas are zeros.

After opening a memory object, the application maps the object into its address space using the *mmap()* function call. Once a mapping has been established, it remains mapped until unmapped with *munmap()*, even if the memory object is closed. The *mprotect()* function can be used to change the memory protections initially established by *mmap()*.

A *close()* of the file descriptor, while invalidating the file descriptor itself, does not unmap any mappings established for the memory object. The address space, including all mapped regions, is inherited on *fork()*. The entire address space is unmapped on process termination or by successful calls to any of the *exec* family of functions.

The *msync()* function is used to force mapped file data to permanent storage.

- Effects on Other Functions

When the Memory Mapped Files option is supported, the operation of the *open()*, *creat()*, and *unlink()* functions are a natural result of using the file system name space to map the global names for memory objects.

The *ftruncate()* function can be used to set the length of a sharable memory object.

The meaning of *stat()* fields other than the size and protection information is undefined on implementations where memory objects are not implemented using regular files. When regular files are used, the times reflect when the implementation updated the file image of the data, not when a process updated the data in memory.

The operations of *fdopen()*, *write()*, *read()*, and *lseek()* were made unspecified for objects opened with *shm\_open()*, so that implementations that did not implement memory objects as regular files would not have to support the operation of these functions on shared memory objects.

The behavior of memory objects with respect to *close()*, *dup()*, *dup2()*, *open()*, *close()*, *fork()*, *\_exit()*, and the *exec* family of functions is the same as the behavior of the existing practice of the *mmap()* function.

A memory object can still be referenced after a close. That is, any mappings made to the file are still in effect, and reads and writes that are made to those mappings are still valid and are shared with other processes that have the same mapping. Likewise, the memory object can still be used if any references remain after its name(s) have been deleted. Any references that remain after a close must not appear to the application as file descriptors.

This is existing practice for *mmap()* and *close()*. In addition, there are already mappings present (text, data, stack) that do not have open file descriptors. The text mapping in particular is considered a reference to the file containing the text. The desire was to treat all mappings by the process uniformly. Also, many modern implementations use *mmap()* to implement shared libraries, and it would not be desirable to keep file descriptors for each of the many libraries an application can use. It was felt there were many other existing programs that used this behavior to free a file descriptor, and thus IEEE Std 1003.1-2001 could not forbid it and still claim to be using existing practice.

For implementations that implement memory objects using memory only, memory objects will retain the memory allocated to the file after the last close and will use that same memory on the next open. Note that closing the memory object is not the same as deleting the name, since the memory object is still defined in the memory object name space.

The locks of *fcntl()* do not block any read or write operation, including read or write access to shared memory or mapped files. In addition, implementations that only support shared memory objects should not be required to implement record locks. The reference to *fcntl()* is added to make this point explicitly. The other *fcntl()* commands are useful with shared memory objects.

The size of pages that mapping hardware may be able to support may be a configurable value, or it may change based on hardware implementations. The addition of the *\_SC\_PAGESIZE* parameter to the *sysconf()* function is provided for determining the mapping page size at runtime.

## Shared Memory Functions

Implementations may support the Shared Memory Objects option without supporting a general Memory Mapped Files option. Shared memory objects are named regions of storage that may be independent of the file system and can be mapped into the address space of one or more processes to allow them to share the associated memory.

- Requirements

Shared memory is used to share data among several processes, each potentially running at different priority levels, responding to different inputs, or performing separate tasks. Shared memory is not just simply providing common access to data, it is providing the fastest possible communication between the processes. With one memory write operation, a process can pass information to as many processes as have the memory region mapped.

As a result, shared memory provides a mechanism that can be used for all other interprocess communication facilities. It may also be used by an application for implementing more

4990 sophisticated mechanisms than semaphores and message queues.

4991 The need for a shared memory interface is obvious for virtual memory systems, where the  
 4992 operating system is directly preventing processes from accessing each other's data. However,  
 4993 in unprotected systems, such as those found in some embedded controllers, a shared  
 4994 memory interface is needed to provide a portable mechanism to allocate a region of memory  
 4995 to be shared and then to communicate the address of that region to other processes.

4996 This, then, provides the minimum functionality that a shared memory interface must have in  
 4997 order to support realtime applications: to allocate and name an object to be mapped into  
 4998 memory for potential sharing (*open()* or *shm\_open()*), and to make the memory object  
 4999 available within the address space of a process (*mmap()*). To complete the interface, a  
 5000 mechanism to release the claim of a process on a shared memory object (*munmap()*) is also  
 5001 needed, as well as a mechanism for deleting the name of a sharable object that was  
 5002 previously created (*unlink()* or *shm\_unlink()*).

5003 After a mapping has been established, an implementation should not have to provide  
 5004 services to maintain that mapping. All memory writes into that area will appear immediately  
 5005 in the memory mapping of that region by any other processes.

5006 Thus, requirements include:

- 5007 — Support creation of sharable memory objects and the mapping of these objects into the  
 5008 address space of a process.
- 5009 — Sharable memory objects should be accessed by global names accessible from all  
 5010 processes.
- 5011 — Support the mapping of specific sections of physical address space (such as a memory  
 5012 mapped device) into the address space of a process. This should not be done by the  
 5013 process specifying the actual address, but again by an implementation-defined global  
 5014 name (such as a special device name) dedicated to this purpose.
- 5015 — Support the mapping of discrete portions of these memory objects.
- 5016 — Support for minimum hardware configurations that contain no physical media on which  
 5017 to store shared memory contents permanently.
- 5018 — The ability to preallocate the entire shared memory region so that minimum hardware  
 5019 configurations without virtual memory support can guarantee contiguous space.
- 5020 — The maximizing of performance by not requiring functionality that would require  
 5021 implementation interaction above creating the shared memory area and returning the  
 5022 mapping.

5023 Note that the above requirements do not preclude:

- 5024 — The sharable memory object from being implemented using actual files on an actual file  
 5025 system.
- 5026 — The global name that is accessible from all processes being restricted to a file system area  
 5027 that is dedicated to handling shared memory.
- 5028 — An implementation not providing implementation-defined global names for the purpose  
 5029 of physical address mapping.

#### 5030 • Shared Memory Objects Usage

5031 If the Shared Memory Objects option is supported, a shared memory object may be created,  
 5032 or opened if it already exists, with the *shm\_open()* function. If the shared memory object is  
 5033 created, it has a length of zero. The *ftruncate()* function can be used to set the size of the

5034 shared memory object after creation. The *shm\_unlink()* function removes the name for a  
 5035 shared memory object created by *shm\_open()*.

5036 • Shared Memory Overview

5037 The shared memory facility defined by IEEE Std 1003.1-2001 usually results in memory  
 5038 locations being added to the address space of the process. The implementation returns the  
 5039 address of the new space to the application by means of a pointer. This works well in  
 5040 languages like C. However, in languages without pointer types it will not work. In the  
 5041 bindings for such a language, either a special COMMON section will need to be defined  
 5042 (which is unlikely), or the binding will have to allow existing structures to be mapped. The  
 5043 implementation will likely have to place restrictions on the size and alignment of such  
 5044 structures or will have to map a suitable region of the address space of the process into the  
 5045 memory object, and thus into other processes. These are issues for that particular language  
 5046 binding. For IEEE Std 1003.1-2001, however, the practice will not be forbidden, merely  
 5047 undefined.

5048 Two potentially different name spaces are used for naming objects that may be mapped into  
 5049 process address spaces. When the Memory Mapped Files option is supported, files may be  
 5050 accessed via *open()*. When the Shared Memory Objects option is supported, sharable  
 5051 memory objects that might not be files may be accessed via the *shm\_open()* function. These  
 5052 options are not mutually-exclusive.

5053 Some implementations supporting the Shared Memory Objects option may choose to  
 5054 implement the shared memory object name space as part of the file system name space.  
 5055 There are several reasons for this:

- 5056 — It allows applications to prevent name conflicts by use of the directory structure.
- 5057 — It uses an existing mechanism for accessing global objects and prevents the creation of a  
 5058 new mechanism for naming global objects.

5059 In such implementations, memory objects can be implemented using regular files, if that is  
 5060 what the implementation chooses. The *shm\_open()* function can be implemented as an *open()*  
 5061 call in a fixed directory followed by a call to *fcntl()* to set FD\_CLOEXEC. The *shm\_unlink()*  
 5062 function can be implemented as an *unlink()* call.

5063 On the other hand, it is also expected that small embedded systems that support the Shared  
 5064 Memory Objects option may wish to implement shared memory without having any file  
 5065 systems present. In this case, the implementations may choose to use a simple string valued  
 5066 name space for shared memory regions. The *shm\_open()* function permits either type of  
 5067 implementation.

5068 Some implementations have hardware that supports protection of mapped data from certain  
 5069 classes of access and some do not. Systems that supply this functionality can support the  
 5070 Memory Protection option.

5071 Some implementations restrict size, alignment, and protections to be on *page-size*  
 5072 boundaries. If an implementation has no restrictions on size or alignment, it may specify a 1-  
 5073 byte page size. Applications on implementations that do support larger pages must be  
 5074 cognizant of the page size since this is the alignment and protection boundary.

5075 Simple embedded implementations may have a 1-byte page size and only support the Shared  
 5076 Memory Objects option. This provides simple shared memory between processes without  
 5077 requiring mapping hardware.

5078 IEEE Std 1003.1-2001 specifically allows a memory object to remain referenced after a close  
 5079 because that is existing practice for the *mmap()* function.

## Typed Memory Functions

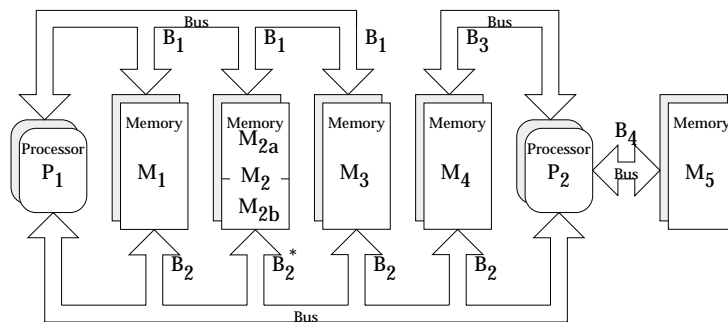
Implementations may support the Typed Memory Objects option without supporting either the Shared Memory option or the Memory Mapped Files option. Typed memory objects are pools of specialized storage, different from the main memory resource normally used by a processor to hold code and data, that can be mapped into the address space of one or more processes.

### • Model

Realtime systems conforming to one of the POSIX.13 realtime profiles are expected (and desired) to be supported on systems with more than one type or pool of memory (for example, SRAM, DRAM, ROM, EPROM, EEPROM), where each type or pool of memory may be accessible by one or more processors via one or more busses (ports). Memory mapped files, shared memory objects, and the language-specific storage allocation operators (*malloc()* for the ISO C standard, *new* for ISO Ada) fail to provide application program interfaces versatile enough to allow applications to control their utilization of such diverse memory resources. The typed memory interfaces *posix\_typed\_mem\_open()*, *posix\_mem\_offset()*, *posix\_typed\_mem\_get\_info()*, *mmap()*, and *munmap()* defined herein support the model of typed memory described below.

For purposes of this model, a system comprises several processors (for example,  $P_1$  and  $P_2$ ), several physical memory pools (for example,  $M_1$ ,  $M_2$ ,  $M_{2a}$ ,  $M_{2b}$ ,  $M_3$ ,  $M_4$ , and  $M_5$ ), and several busses or “ports” (for example,  $B_1$ ,  $B_2$ ,  $B_3$ , and  $B_4$ ) interconnecting the various processors and memory pools in some system-specific way. Notice that some memory pools may be contained in others (for example,  $M_{2a}$  and  $M_{2b}$  are contained in  $M_2$ ).

Figure B-1 shows an example of such a model. In a system like this, an application should be able to perform the following operations:



\* All addresses in pool  $M_2$  (comprising pools  $M_{2a}$  and  $M_{2b}$ ) accessible via port  $B_1$ .  
 Addresses in pool  $M_{2b}$  are also accessible via port  $B_2$ .  
 Addresses in pool  $M_{2a}$  are *not* accessible via port  $B_2$ .

**Figure B-1** Example of a System with Typed Memory

### — Typed Memory Allocation

An application should be able to allocate memory dynamically from the desired pool using the desired bus, and map it into a process' address space. For example, processor  $P_1$  can allocate some portion of memory pool  $M_1$  through port  $B_1$ , treating all unmapped subareas of  $M_1$  as a heap-storage resource from which memory may be allocated. This portion of memory is mapped into the process' address space, and subsequently deallocated when unmapped from all processes.



— Using the Same Storage Region from Different Busses

An application process with a mapped region of storage that is accessed from one bus should be able to map that same storage area at another address (subject to page size restrictions detailed in *mmap()*), to allow it to be accessed from another bus. For example, processor  $P_1$  may wish to access the same region of memory pool  $M_{2b}$  both through ports  $B_1$  and  $B_2$ .

— Sharing Typed Memory Regions

Several application processes running on the same or different processors may wish to share a particular region of a typed memory pool. Each process or processor may wish to access this region through different busses. For example, processor  $P_1$  may want to share a region of memory pool  $M_4$  with processor  $P_2$ , and they may be required to use busses  $B_2$  and  $B_3$ , respectively, to minimize bus contention. A problem arises here when a process allocates and maps a portion of fragmented memory and then wants to share this region of memory with another process, either in the same processor or different processors. The solution adopted is to allow the first process to find out the memory map (offsets and lengths) of all the different fragments of memory that were mapped into its address space, by repeatedly calling *posix\_mem\_offset()*. Then, this process can pass the offsets and lengths obtained to the second process, which can then map the same memory fragments into its address space.

— Contiguous Allocation

The problem of finding the memory map of the different fragments of the memory pool that were mapped into logically contiguous addresses of a given process can be solved by requesting contiguous allocation. For example, a process in  $P_1$  can allocate 10 Kbytes of physically contiguous memory from  $M_3-B_1$ , and obtain the offset (within pool  $M_3$ ) of this block of memory. Then, it can pass this offset (and the length) to a process in  $P_2$  using some interprocess communication mechanism. The second process can map the same block of memory by using the offset transferred and specifying  $M_3-B_2$ .

— Unallocated Mapping

Any subarea of a memory pool that is mapped to a process, either as the result of an allocation request or an explicit mapping, is normally unavailable for allocation. Special processes such as debuggers, however, may need to map large areas of a typed memory pool, yet leave those areas available for allocation.

Typed memory allocation and mapping has to coexist with storage allocation operators like *malloc()*, but systems are free to choose how to implement this coexistence. For example, it may be system configuration-dependent if all available system memory is made part of one of the typed memory pools or if some part will be restricted to conventional allocation operators. Equally system configuration-dependent may be the availability of operators like *malloc()* to allocate storage from certain typed memory pools. It is not excluded to configure a system such that a given named pool,  $P_1$ , is in turn split into non-overlapping named subpools. For example,  $M_1-B_1$ ,  $M_2-B_1$ , and  $M_3-B_1$  could also be accessed as one common pool  $M_{123}-B_1$ . A call to *malloc()* on  $P_1$  could work on such a larger pool while full optimization of memory usage by  $P_1$  would require typed memory allocation at the subpool level.

• Existing Practice

OS-9 provides for the naming (numbering) and prioritization of memory types by a system administrator. It then provides APIs to request memory allocation of typed (colored) memory by number, and to generate a bus address from a mapped memory address (translate). When requesting colored memory, the user can specify type 0 to signify allocation

from the first available type in priority order.

HP-RT presents interfaces to map different kinds of storage regions that are visible through a VME bus, although it does not provide allocation operations. It also provides functions to perform address translation between VME addresses and virtual addresses. It represents a VME-bus unique solution to the general problem.

The PSOS approach is similar (that is, based on a pre-established mapping of bus address ranges to specific memories) with a concept of segments and regions (regions dynamically allocated from a heap which is a special segment). Therefore, PSOS does not fully address the general allocation problem either. PSOS does not have a “process”-based model, but more of a “thread”-only-based model of multi-tasking. So mapping to a process address space is not an issue.

QNX uses the System V approach of opening specially named devices (shared memory segments) and using *mmap()* to then gain access from the process. They do not address allocation directly, but once typed shared memory can be mapped, an “allocation manager” process could be written to handle requests for allocation.

The System V approach also included allocation, implemented by opening yet other special “devices” which allocate, rather than appearing as a whole memory object.

The Orkid realtime kernel interface definition has operations to manage memory “regions” and “pools”, which are areas of memory that may reflect the differing physical nature of the memory. Operations to allocate memory from these regions and pools are also provided.

#### • Requirements

Existing practice in SVID-derived UNIX systems relies on functionality similar to *mmap()* and its related interfaces to achieve mapping and allocation of typed memory. However, the issue of sharing typed memory (allocated or mapped) and the complication of multiple ports are not addressed in any consistent way by existing UNIX system practice. Part of this functionality is existing practice in specialized realtime operating systems. In order to solidify the capabilities implied by the model above, the following requirements are imposed on the interface:

##### — Identification of Typed Memory Pools and Ports

All processes (running in all processors) in the system are able to identify a particular (system configured) typed memory pool accessed through a particular (system configured) port by a name. That name is a member of a name space common to all these processes, but need not be the same name space as that containing ordinary filenames. The association between memory pools/ports and corresponding names is typically established when the system is configured. The “open” operation for typed memory objects should be distinct from the *open()* function, for consistency with other similar services, but implementable on top of *open()*. This implies that the handle for a typed memory object will be a file descriptor.

##### — Allocation and Mapping of Typed Memory

Once a typed memory object has been identified by a process, it is possible to both map user-selected subareas of that object into process address space and to map system-selected (that is, dynamically allocated) subareas of that object, with user-specified length, into process address space. It is also possible to determine the maximum length of memory allocation that may be requested from a given typed memory object.

1

- 5203 — Sharing Typed Memory
- 5204 Two or more processes are able to share portions of typed memory, either user-selected or  
5205 dynamically allocated. This requirement applies also to dynamically allocated regions of  
5206 memory that are composed of several non-contiguous pieces.
- 5207 — Contiguous Allocation
- 5208 For dynamic allocation, it is the user's option whether the system is required to allocate a  
5209 contiguous subarea within the typed memory object, or whether it is permitted to allocate  
5210 discontinuous fragments which appear contiguous in the process mapping. Contiguous  
5211 allocation simplifies the process of sharing allocated typed memory, while discontinuous  
5212 allocation allows for potentially better recovery of deallocated typed memory.
- 5213 — Accessing Typed Memory Through Different Ports
- 5214 Once a subarea of a typed memory object has been mapped, it is possible to determine the  
5215 location and length corresponding to a user-selected portion of that object within the  
5216 memory pool. This location and length can then be used to remap that portion of memory  
5217 for access from another port. If the referenced portion of typed memory was allocated  
5218 discontinuously, the length thus determined may be shorter than anticipated, and the  
5219 user code must adapt to the value returned.
- 5220 — Deallocation
- 5221 When a previously mapped subarea of typed memory is no longer mapped by any  
5222 process in the system—as a result of a call or calls to *munmap()*—that subarea becomes  
5223 potentially reusable for dynamic allocation; actual reuse of the subarea is a function of the  
5224 dynamic typed memory allocation policy.
- 5225 — Unallocated Mapping
- 5226 It must be possible to map user-selected subareas of a typed memory object without  
5227 marking that subarea as unavailable for allocation. This option is not the default behavior,  
5228 and requires appropriate privilege.
- 5229 • Scenario
- 5230 The following scenario will serve to clarify the use of the typed memory interfaces.
- 5231 Process A running on  $P_1$  (see Figure B-1 (on page 124)) wants to allocate some memory from  
5232 memory pool  $M_2$ , and it wants to share this portion of memory with process B running on  $P_2$ .  
5233 Since  $P_2$  only has access to the lower part of  $M_2$ , both processes will use the memory pool  
5234 named  $M_{2b}$  which is the part of  $M_2$  that is accessible both from  $P_1$  and  $P_2$ . The operations that  
5235 both processes need to perform are shown below:
- 5236 — Allocating Typed Memory
- 5237 Process A calls *posix\_typed\_mem\_open()* with the name **/typed.m2b-b1** and a *tflag* of  
5238 **POSIX\_TYPED\_MEM\_ALLOCATE** to get a file descriptor usable for allocating from pool  
5239  $M_{2b}$  accessed through port  $B_1$ . It then calls *mmap()* with this file descriptor requesting a  
5240 length of 4096 bytes. The system allocates two discontinuous blocks of sizes 1024 and  
5241 3072 bytes within  $M_{2b}$ . The *mmap()* function returns a pointer to a 4096-byte array in  
5242 process A's logical address space, mapping the allocated blocks contiguously. Process A  
5243 can then utilize the array, and store data in it.
- 5244 — Determining the Location of the Allocated Blocks
- 5245 Process A can determine the lengths and offsets (relative to  $M_{2b}$ ) of the two blocks  
5246 allocated, by using the following procedure: First, process A calls *posix\_mem\_offset()* with

the address of the first element of the array and length 4 096. Upon return, the offset and length (1 024 bytes) of the first block are returned. A second call to *posix\_mem\_offset()* is then made using the address of the first element of the array plus 1 024 (the length of the first block), and a new length of 4 096–1 024. If there were more fragments allocated, this procedure could have been continued within a loop until the offsets and lengths of all the blocks were obtained. Notice that this relatively complex procedure can be avoided if contiguous allocation is requested (by opening the typed memory object with the *tflag* `POSIX_TYPED_MEM_ALLOCATE_CONTIG`).

#### — Sharing Data Across Processes

Process A passes the two offset values and lengths obtained from the *posix\_mem\_offset()* calls to process B running on  $P_2$ , via some form of interprocess communication. Process B can gain access to process A's data by calling *posix\_typed\_mem\_open()* with the name `/typed.m2b-b2` and a *tflag* of zero, then using two *mmap()* calls on the resulting file descriptor to map the two subareas of that typed memory object to its own address space.

#### • Rationale for no *mem\_alloc()* and *mem\_free()*

The standard developers had originally proposed a pair of new flags to *mmap()* which, when applied to a typed memory object descriptor, would cause *mmap()* to allocate dynamically from an unallocated and unmapped area of the typed memory object. Deallocation was similarly accomplished through the use of *munmap()*. This was rejected by the ballot group because it excessively complicated the (already rather complex) *mmap()* interface and introduced semantics useful only for typed memory, to a function which must also map shared memory and files. They felt that a memory allocator should be built on top of *mmap()* instead of being incorporated within the same interface, much as the ISO C standard libraries build *malloc()* on top of the virtual memory mapping functions *brk()* and *sbrk()*. This would eliminate the complicated semantics involved with unmapping only part of an allocated block of typed memory.

To attempt to achieve ballot group consensus, typed memory allocation and deallocation was first migrated from *mmap()* and *munmap()* to a pair of complementary functions modeled on the ISO C standard *malloc()* and *free()*. The *mem\_alloc()* function specified explicitly the typed memory object (typed memory pool/access port) from which allocation takes place, unlike *malloc()* where the memory pool and port are unspecified. The *mem\_free()* function handled deallocation. These new semantics still met all of the requirements detailed above without modifying the behavior of *mmap()* except to allow it to map specified areas of typed memory objects. An implementation would have been free to implement *mem\_alloc()* and *mem\_free()* over *mmap()*, through *mmap()*, or independently but cooperating with *mmap()*.

The ballot group was queried to see if this was an acceptable alternative, and while there was some agreement that it achieved the goal of removing the complicated semantics of allocation from the *mmap()* interface, several balloters realized that it just created two additional functions that behaved, in great part, like *mmap()*. These balloters proposed an alternative which has been implemented here in place of a separate *mem\_alloc()* and *mem\_free()*. This alternative is based on four specific suggestions:

1. The *posix\_typed\_mem\_open()* function should provide a flag which specifies “allocate on *mmap()*” (otherwise, *mmap()* just maps the underlying object). This allows things roughly similar to `/dev/zero` versus `/dev/swap`. Two such flags have been implemented, one of which forces contiguous allocation.
2. The *posix\_mem\_offset()* function is acceptable because it can be applied usefully to mapped objects in general. It should return the file descriptor of the underlying object.

3. The `mem_get_info()` function in an earlier draft should be renamed `posix_typed_mem_get_info()` because it is not generally applicable to memory objects. It should probably return the file descriptor's allocation attribute. The renaming of the function has been implemented, but having it return a piece of information which is readily known by an application without this function has been rejected. Its whole purpose is to query the typed memory object for attributes that are not user-specified, but determined by the implementation.

4. There should be no separate `mem_alloc()` or `mem_free()` functions. Instead, using `mmap()` on a typed memory object opened with an "allocate on `mmap()`" flag should be used to force allocation. These are precisely the semantics defined in the current draft.

- Rationale for no Typed Memory Access Management

The working group had originally defined an additional interface (and an additional kind of object: typed memory master) to establish and dissolve mappings to typed memory on behalf of devices or processors which were independent of the operating system and had no inherent capability to directly establish mappings on their own. This was to have provided functionality similar to device driver interfaces such as `physio()` and their underlying bus-specific interfaces (for example, `mballoc()`) which serve to set up and break down DMA pathways, and derive mapped addresses for use by hardware devices and processor cards.

The ballot group felt that this was beyond the scope of POSIX.1 and its amendments. Furthermore, the removal of interrupt handling interfaces from a preceding amendment (the IEEE Std 1003.1d-1999) during its balloting process renders these typed memory access management interfaces an incomplete solution to portable device management from a user process; it would be possible to initiate a device transfer to/from typed memory, but impossible to handle the transfer-complete interrupt in a portable way.

To achieve ballot group consensus, all references to typed memory access management capabilities were removed. The concept of portable interfaces from a device driver to both operating system and hardware is being addressed by the Uniform Driver Interface (UDI) industry forum, with formal standardization deferred until proof of concept and industry-wide acceptance and implementation.

#### B.2.8.4 Process Scheduling

IEEE PASC Interpretation 1003.1 #96 has been applied, adding the `pthread_setschedprio()` function. This was added since previously there was no way for a thread to lower its own priority without going to the tail of the threads list for its new priority. This capability is necessary to bound the duration of priority inversion encountered by a thread.

The following portion of the rationale presents models, requirements, and standardization issues relevant to process scheduling; see also Section B.2.9.4 (on page 169).

In an operating system supporting multiple concurrent processes, the system determines the order in which processes execute to meet implementation-defined goals. For time-sharing systems, the goal is to enhance system throughput and promote fairness; the application is provided with little or no control over this sequencing function. While this is acceptable and desirable behavior in a time-sharing system, it is inappropriate in a realtime system; realtime applications must specifically control the execution sequence of their concurrent processes in order to meet externally defined response requirements.

In IEEE Std 1003.1-2001, the control over process sequencing is provided using a concept of scheduling policies. These policies, described in detail in this section, define the behavior of the system whenever processor resources are to be allocated to competing processes. Only the behavior of the policy is defined; conforming implementations are free to use any mechanism

desired to achieve the described behavior.

- Models

In an operating system supporting multiple concurrent processes, the system determines the order in which processes execute and might force long-running processes to yield to other processes at certain intervals. Typically, the scheduling code is executed whenever an event occurs that might alter the process to be executed next.

The simplest scheduling strategy is a “first-in, first-out” (FIFO) dispatcher. Whenever a process becomes runnable, it is placed on the end of a ready list. The process at the front of the ready list is executed until it exits or becomes blocked, at which point it is removed from the list. This scheduling technique is also known as “run-to-completion” or “run-to-block”.

A natural extension to this scheduling technique is the assignment of a “non-migrating priority” to each process. This policy differs from strict FIFO scheduling in only one respect: whenever a process becomes runnable, it is placed at the end of the list of processes runnable at that priority level. When selecting a process to run, the system always selects the first process from the highest priority queue with a runnable process. Thus, when a process becomes unblocked, it will preempt a running process of lower priority without otherwise altering the ready list. Further, if a process elects to alter its priority, it is removed from the ready list and reinserted, using its new priority, according to the policy above.

While the above policy might be considered unfriendly in a time-sharing environment in which multiple users require more balanced resource allocation, it could be ideal in a realtime environment for several reasons. The most important of these is that it is deterministic: the highest-priority process is always run and, among processes of equal priority, the process that has been runnable for the longest time is executed first. Because of this determinism, cooperating processes can implement more complex scheduling simply by altering their priority. For instance, if processes at a single priority were to reschedule themselves at fixed time intervals, a time-slice policy would result.

In a dedicated operating system in which all processes are well-behaved realtime applications, non-migrating priority scheduling is sufficient. However, many existing implementations provide for more complex scheduling policies.

IEEE Std 1003.1-2001 specifies a linear scheduling model. In this model, every process in the system has a priority. The system scheduler always dispatches a process that has the highest (generally the most time-critical) priority among all runnable processes in the system. As long as there is only one such process, the dispatching policy is trivial. When multiple processes of equal priority are eligible to run, they are ordered according to a strict run-to-completion (FIFO) policy.

The priority is represented as a positive integer and is inherited from the parent process. For processes running under a fixed priority scheduling policy, the priority is never altered except by an explicit function call.

It was determined arbitrarily that larger integers correspond to “higher priorities”.

Certain implementations might impose restrictions on the priority ranges to which processes can be assigned. There also can be restrictions on the set of policies to which processes can be set.

- Requirements

Realtime processes require that scheduling be fast and deterministic, and that it guarantees to preempt lower priority processes.

Thus, given the linear scheduling model, realtime processes require that they be run at a priority that is higher than other processes. Within this framework, realtime processes are free to yield execution resources to each other in a completely portable and implementation-defined manner.

As there is a generally perceived requirement for processes at the same priority level to share processor resources more equitably, provisions are made by providing a scheduling policy (that is, SCHED\_RR) intended to provide a timeslice-like facility.

**Note:** The following topics assume that low numeric priority implies low scheduling criticality and *vice versa*.

- Rationale for New Interface

Realtime applications need to be able to determine when processes will run in relation to each other. It must be possible to guarantee that a critical process will run whenever it is runnable; that is, whenever it wants to for as long as it needs. SCHED\_FIFO satisfies this requirement. Additionally, SCHED\_RR was defined to meet a realtime requirement for a well-defined time-sharing policy for processes at the same priority.

It would be possible to use the BSD *setpriority()* and *getpriority()* functions by redefining the meaning of the “nice” parameter according to the scheduling policy currently in use by the process. The System V *nice()* interface was felt to be undesirable for realtime because it specifies an adjustment to the “nice” value, rather than setting it to an explicit value. Realtime applications will usually want to set priority to an explicit value. Also, System V *nice()* does not allow for changing the priority of another process.

With the POSIX.1b interfaces, the traditional “nice” value does not affect the SCHED\_FIFO or SCHED\_RR scheduling policies. If a “nice” value is supported, it is implementation-defined whether it affects the SCHED\_OTHER policy.

An important aspect of IEEE Std 1003.1-2001 is the explicit description of the queuing and preemption rules. It is critical, to achieve deterministic scheduling, that such rules be stated clearly in IEEE Std 1003.1-2001.

IEEE Std 1003.1-2001 does not address the interaction between priority and swapping. The issues involved with swapping and virtual memory paging are extremely implementation-defined and would be nearly impossible to standardize at this point. The proposed scheduling paradigm, however, fully describes the scheduling behavior of runnable processes, of which one criterion is that the working set be resident in memory. Assuming the existence of a portable interface for locking portions of a process in memory, paging behavior need not affect the scheduling of realtime processes.

IEEE Std 1003.1-2001 also does not address the priorities of “system” processes. In general, these processes should always execute in low-priority ranges to avoid conflict with other realtime processes. Implementations should document the priority ranges in which system processes run.

The default scheduling policy is not defined. The effect of I/O interrupts and other system processing activities is not defined. The temporary lending of priority from one process to another (such as for the purposes of affecting freeing resources) by the system is not addressed. Preemption of resources is not addressed. Restrictions on the ability of a process to affect other processes beyond a certain level (influence levels) is not addressed.

The rationale used to justify the simple time-quantum scheduler is that it is common practice to depend upon this type of scheduling to ensure “fair” distribution of processor resources among portions of the application that must interoperate in a serial fashion. Note that IEEE Std 1003.1-2001 is silent with respect to the setting of this time quantum, or whether it is

a system-wide value or a per-process value, although it appears that the prevailing realtime practice is for it to be a system-wide value.

In a system with  $N$  processes at a given priority, all processor-bound, in which the time quantum is equal for all processes at a specific priority level, the following assumptions are made of such a scheduling policy:

1. A time quantum  $Q$  exists and the current process will own control of the processor for at least a duration of  $Q$  and will have the processor for a duration of  $Q$ .
2. The  $N$ th process at that priority will control a processor within a duration of  $(N-1) \times Q$ .

These assumptions are necessary to provide equal access to the processor and bounded response from the application.

The assumptions hold for the described scheduling policy only if no system overhead, such as interrupt servicing, is present. If the interrupt servicing load is non-zero, then one of the two assumptions becomes fallacious, based upon how  $Q$  is measured by the system.

If  $Q$  is measured by clock time, then the assumption that the process obtains a duration  $Q$  processor time is false if interrupt overhead exists. Indeed, a scenario can be constructed with  $N$  processes in which a single process undergoes complete processor starvation if a peripheral device, such as an analog-to-digital converter, generates significant interrupt activity periodically with a period of  $N \times Q$ .

If  $Q$  is measured as actual processor time, then the assumption that the  $N$ th process runs in within the duration  $(N-1) \times Q$  is false.

It should be noted that SCHED\_FIFO suffers from interrupt-based delay as well. However, for SCHED\_FIFO, the implied response of the system is “as soon as possible”, so that the interrupt load for this case is a vendor selection and not a compliance issue.

With this in mind, it is necessary either to complete the definition by including bounds on the interrupt load, or to modify the assumptions that can be made about the scheduling policy.

Since the motivation of inclusion of the policy is common usage, and since current applications do not enjoy the luxury of bounded interrupt load, item (2) above is sufficient to express existing application needs and is less restrictive in the standard definition. No difference in interface is necessary.

In an implementation in which the time quantum is equal for all processes at a specific priority, our assumptions can then be restated as:

- A time quantum  $Q$  exists, and a processor-bound process will be rescheduled after a duration of, at most,  $Q$ . Time quantum  $Q$  may be defined in either wall clock time or execution time.
- In general, the  $N$ th process of a priority level should wait no longer than  $(N-1) \times Q$  time to execute, assuming no processes exist at higher priority levels.
- No process should wait indefinitely.

For implementations supporting per-process time quanta, these assumptions can be readily extended.



## Sporadic Server Scheduling Policy

The sporadic server is a mechanism defined for scheduling aperiodic activities in time-critical realtime systems. This mechanism reserves a certain bounded amount of execution capacity for processing aperiodic events at a high priority level. Any aperiodic events that cannot be processed within the bounded amount of execution capacity are executed in the background at a low priority level. Thus, a certain amount of execution capacity can be guaranteed to be available for processing periodic tasks, even under burst conditions in the arrival of aperiodic processing requests (that is, a large number of requests in a short time interval). The sporadic server also simplifies the schedulability analysis of the realtime system, because it allows aperiodic processes or threads to be treated as if they were periodic. The sporadic server was first described by Sprunt, et al.

The key concept of the sporadic server is to provide and limit a certain amount of computation capacity for processing aperiodic events at their assigned normal priority, during a time interval called the “replenishment period”. Once the entity controlled by the sporadic server mechanism is initialized with its period and execution-time budget attributes, it preserves its execution capacity until an aperiodic request arrives. The request will be serviced (if there are no higher priority activities pending) as long as there is execution capacity left. If the request is completed, the actual execution time used to service it is subtracted from the capacity, and a replenishment of this amount of execution time is scheduled to happen one replenishment period after the arrival of the aperiodic request. If the request is not completed, because there is no execution capacity left, then the aperiodic process or thread is assigned a lower background priority. For each portion of consumed execution capacity the execution time used is replenished after one replenishment period. At the time of replenishment, if the sporadic server was executing at a background priority level, its priority is elevated to the normal level. Other similar replenishment policies have been defined, but the one presented here represents a compromise between efficiency and implementation complexity.

The interface that appears in this section defines a new scheduling policy for threads and processes that behaves according to the rules of the sporadic server mechanism. Scheduling attributes are defined and functions are provided to allow the user to set and get the parameters that control the scheduling behavior of this mechanism, namely the normal and low priority, the replenishment period, the maximum number of pending replenishment operations, and the initial execution-time budget.

- Scheduling Aperiodic Activities

Virtually all realtime applications are required to process aperiodic activities. In many cases, there are tight timing constraints that the response to the aperiodic events must meet. Usual timing requirements imposed on the response to these events are:

- The effects of an aperiodic activity on the response time of lower priority activities must be controllable and predictable.
- The system must provide the fastest possible response time to aperiodic events.
- It must be possible to take advantage of all the available processing bandwidth not needed by time-critical activities to enhance average-case response times to aperiodic events.

Traditional methods for scheduling aperiodic activities are background processing, polling tasks, and direct event execution:

- Background processing consists of assigning a very low priority to the processing of aperiodic events. It utilizes all the available bandwidth in the system that has not been consumed by higher priority threads. However, it is very difficult, or impossible, to meet

- 5519 requirements on average-case response time, because the aperiodic entity has to wait for  
5520 the execution of all other entities which have higher priority.
- 5521 — Polling consists of creating a periodic process or thread for servicing aperiodic requests.  
5522 At regular intervals, the polling entity is started and its services accumulated pending  
5523 aperiodic requests. If no aperiodic requests are pending, the polling entity suspends itself  
5524 until its next period. Polling allows the aperiodic requests to be processed at a higher  
5525 priority level. However, worst and average-case response times of polling entities are a  
5526 direct function of the polling period, and there is execution overhead for each polling  
5527 period, even if no event has arrived. If the deadline of the aperiodic activity is short  
5528 compared to the inter-arrival time, the polling frequency must be increased to guarantee  
5529 meeting the deadline. For this case, the increase in frequency can dramatically reduce the  
5530 efficiency of the system and, therefore, its capacity to meet all deadlines. Yet, polling  
5531 represents a good way to handle a large class of practical problems because it preserves  
5532 system predictability, and because the amortized overhead drops as load increases.
  - 5533 — Direct event execution consists of executing the aperiodic events at a high fixed-priority  
5534 level. Typically, the aperiodic event is processed by an interrupt service routine as soon as  
5535 it arrives. This technique provides predictable response times for aperiodic events, but  
5536 makes the response times of all lower priority activities completely unpredictable under  
5537 burst arrival conditions. Therefore, if the density of aperiodic event arrivals is  
5538 unbounded, it may be a dangerous technique for time-critical systems. Yet, for those cases  
5539 in which the physics of the system imposes a bound on the event arrival rate, it is  
5540 probably the most efficient technique.
  - 5541 — The sporadic server scheduling algorithm combines the predictability of the polling  
5542 approach with the short response times of the direct event execution. Thus, it allows  
5543 systems to meet an important class of application requirements that cannot be met by  
5544 using the traditional approaches. Multiple sporadic servers with different attributes can  
5545 be applied to the scheduling of multiple classes of aperiodic events, each with different  
5546 kinds of timing requirements, such as individual deadlines, average response times, and  
5547 so on. It also has many other interesting applications for realtime, such as scheduling  
5548 producer/consumer tasks in time-critical systems, limiting the effects of faults on the  
5549 estimation of task execution-time requirements, and so on.
- 5550 • Existing Practice
- 5551 The sporadic server has been used in different kinds of applications, including military  
5552 avionics, robot control systems, industrial automation systems, and so on. There are  
5553 examples of many systems that cannot be successfully scheduled using the classic  
5554 approaches, such as direct event execution, or polling, and are schedulable using a sporadic  
5555 server scheduler. The sporadic server algorithm itself can successfully schedule all systems  
5556 scheduled with direct event execution or polling.
- 5557 The sporadic server scheduling policy has been implemented as a commercial product in the  
5558 run-time system of the Verdex Ada compiler. There are also many applications that have  
5559 used a much less efficient application-level sporadic server. These realtime applications  
5560 would benefit from a sporadic server scheduler implemented at the scheduler level.
- 5561 • Library-Level *versus* Kernel-Level Implementation
- 5562 The sporadic server interface described in this section requires the sporadic server policy to  
5563 be implemented at the same level as the scheduler. This means that the process sporadic  
5564 server must be implemented at the kernel level and the thread sporadic server policy  
5565 implemented at the same level as the thread scheduler; that is, kernel or library level.

In an earlier interface for the sporadic server, this mechanism was implementable at a different level than the scheduler. This feature allowed the implementor to choose between an efficient scheduler-level implementation, or a simpler user or library-level implementation. However, the working group considered that this interface made the use of sporadic servers more complex, and that library-level implementations would lack some of the important functionality of the sporadic server, namely the limitation of the actual execution time of aperiodic activities. The working group also felt that the interface described in this chapter does not preclude library-level implementations of threads intended to provide efficient low-overhead scheduling for those threads that are not scheduled under the sporadic server policy.

- Range of Scheduling Priorities

Each of the scheduling policies supported in IEEE Std 1003.1-2001 has an associated range of priorities. The priority ranges for each policy might or might not overlap with the priority ranges of other policies. For time-critical realtime applications it is usual for periodic and aperiodic activities to be scheduled together in the same processor. Periodic activities will usually be scheduled using the SCHED\_FIFO scheduling policy, while aperiodic activities may be scheduled using SCHED\_SPORADIC. Since the application developer will require complete control over the relative priorities of these activities in order to meet his timing requirements, it would be desirable for the priority ranges of SCHED\_FIFO and SCHED\_SPORADIC to overlap completely. Therefore, although IEEE Std 1003.1-2001 does not require any particular relationship between the different priority ranges, it is recommended that these two ranges should coincide.

- Dynamically Setting the Sporadic Server Policy

Several members of the working group requested that implementations should not be required to support dynamically setting the sporadic server scheduling policy for a thread. The reason is that this policy may have a high overhead for library-level implementations of threads, and if threads are allowed to dynamically set this policy, this overhead can be experienced even if the thread does not use that policy. By disallowing the dynamic setting of the sporadic server scheduling policy, these implementations can accomplish efficient scheduling for threads using other policies. If a strictly conforming application needs to use the sporadic server policy, and is therefore willing to pay the overhead, it must set this policy at the time of thread creation.

- Limitation of the Number of Pending Replenishments

The number of simultaneously pending replenishment operations must be limited for each sporadic server for two reasons: an unlimited number of replenishment operations would need an unlimited number of system resources to store all the pending replenishment operations; on the other hand, in some implementations each replenishment operation will represent a source of priority inversion (just for the duration of the replenishment operation) and thus, the maximum amount of replenishments must be bounded to guarantee bounded response times. The way in which the number of replenishments is bounded is by lowering the priority of the sporadic server to *sched\_ss\_low\_priority* when the number of pending replenishments has reached its limit. In this way, no new replenishments are scheduled until the number of pending replenishments decreases.

In the sporadic server scheduling policy defined in IEEE Std 1003.1-2001, the application can specify the maximum number of pending replenishment operations for a single sporadic server, by setting the value of the *sched\_ss\_max\_repl* scheduling parameter. This value must be between one and {SS\_REPL\_MAX}, which is a maximum limit imposed by the implementation. The limit {SS\_REPL\_MAX} must be greater than or equal to {\_POSIX\_SS\_REPL\_MAX}, which is defined to be four in IEEE Std 1003.1-2001. The minimum

5615 limit of four was chosen so that an application can at least guarantee that four different  
 5616 aperiodic events can be processed during each interval of length equal to the replenishment  
 5617 period.

#### 5618 B.2.8.5 Clocks and Timers

##### 5619 • Clocks

5620 IEEE Std 1003.1-2001 and the ISO C standard both define functions for obtaining system time.  
 5621 Implicit behind these functions is a mechanism for measuring passage of time. This  
 5622 specification makes this mechanism explicit and calls it a clock. The CLOCK\_REALTIME  
 5623 clock required by IEEE Std 1003.1-2001 is a higher resolution version of the clock that  
 5624 maintains POSIX.1 system time. This is a “system-wide” clock, in that it is visible to all  
 5625 processes and, were it possible for multiple processes to all read the clock at the same time,  
 5626 they would see the same value.

5627 An extensible interface was defined, with the ability for implementations to define additional  
 5628 clocks. This was done because of the observation that many realtime platforms support  
 5629 multiple clocks, and it was desired to fit this model within the standard interface. But  
 5630 implementation-defined clocks need not represent actual hardware devices, nor are they  
 5631 necessarily system-wide.

##### 5632 • Timers

5633 Two timer types are required for a system to support realtime applications:

##### 5634 1. One-shot

5635 A one-shot timer is a timer that is armed with an initial expiration time, either relative  
 5636 to the current time or at an absolute time (based on some timing base, such as time in  
 5637 seconds and nanoseconds since the Epoch). The timer expires once and then is  
 5638 disarmed. With the specified facilities, this is accomplished by setting the *it\_value*  
 5639 member of the *value* argument to the desired expiration time and the *it\_interval* member  
 5640 to zero.

##### 5641 2. Periodic

5642 A periodic timer is a timer that is armed with an initial expiration time, again either  
 5643 relative or absolute, and a repetition interval. When the initial expiration occurs, the  
 5644 timer is reloaded with the repetition interval and continues counting. With the  
 5645 specified facilities, this is accomplished by setting the *it\_value* member of the *value*  
 5646 argument to the desired initial expiration time and the *it\_interval* member to the desired  
 5647 repetition interval.

5648 For both of these types of timers, the time of the initial timer expiration can be specified in  
 5649 two ways:

##### 5650 1. Relative (to the current time)

##### 5651 2. Absolute

##### 5652 • Examples of Using Realtime Timers

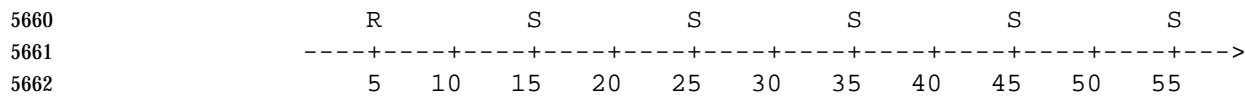
5653 In the diagrams below, *S* indicates a program schedule, *R* shows a schedule method request,  
 5654 and *E* suggests an internal operating system event.

##### 5655 — Periodic Timer: Data Logging

5656 During an experiment, it might be necessary to log realtime data periodically to an  
 5657 internal buffer or to a mass storage device. With a periodic scheduling method, a logging

5658 module can be started automatically at fixed time intervals to log the data.

5659 Program schedule is requested every 10 seconds.



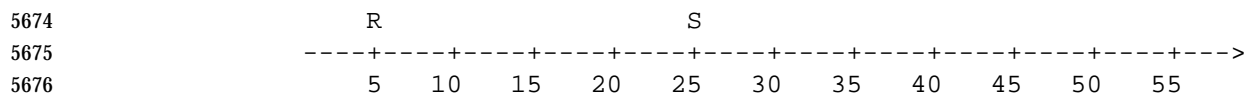
5663 [Time (in Seconds)]

5664 To achieve this type of scheduling using the specified facilities, one would allocate a per-  
 5665 process timer based on clock ID `CLOCK_REALTIME`. Then the timer would be armed via  
 5666 a call to `timer_settime()` with the `TIMER_ABSTIME` flag reset, and with an initial  
 5667 expiration value and a repetition interval of 10 seconds.

5668 — One-shot Timer (Relative Time): Device Initialization

5669 In an emission test environment, large sample bags are used to capture the exhaust from  
 5670 a vehicle. The exhaust is purged from these bags before each and every test. With a one-  
 5671 shot timer, a module could initiate the purge function and then suspend itself for a  
 5672 predetermined period of time while the sample bags are prepared.

5673 Program schedule requested 20 seconds after call is issued.



5677 [Time (in Seconds)]

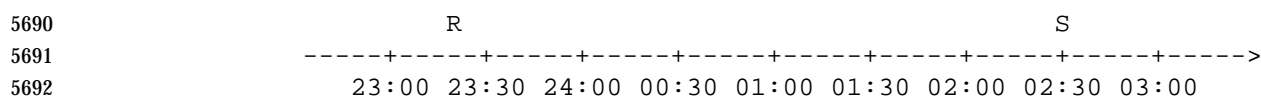
5678 To achieve this type of scheduling using the specified facilities, one would allocate a per-  
 5679 process timer based on clock ID `CLOCK_REALTIME`. Then the timer would be armed via  
 5680 a call to `timer_settime()` with the `TIMER_ABSTIME` flag reset, and with an initial  
 5681 expiration value of 20 seconds and a repetition interval of zero.

5682 Note that if the program wishes merely to suspend itself for the specified interval, it  
 5683 could more easily use `nanosleep()`.

5684 — One-shot Timer (Absolute Time): Data Transmission

5685 The results from an experiment are often moved to a different system within a network  
 5686 for postprocessing or archiving. With an absolute one-shot timer, a module that moves  
 5687 data from a test-cell computer to a host computer can be automatically scheduled on a  
 5688 daily basis.

5689 Program schedule requested for 2:30 a.m.



5693 [Time of Day]

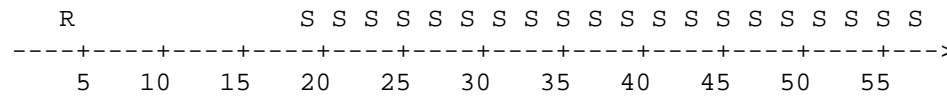
5694 To achieve this type of scheduling using the specified facilities, a per-process timer would  
 5695 be allocated based on clock ID `CLOCK_REALTIME`. Then the timer would be armed via  
 5696 a call to `timer_settime()` with the `TIMER_ABSTIME` flag set, and an initial expiration value  
 5697 equal to 2:30 a.m. of the next day.

5698 — Periodic Timer (Relative Time): Signal Stabilization

5699 Some measurement devices, such as emission analyzers, do not respond instantaneously  
 5700 to an introduced sample. With a periodic timer with a relative initial expiration time, a

module that introduces a sample and records the average response could suspend itself for a predetermined period of time while the signal is stabilized and then sample at a fixed rate.

Program schedule requested 15 seconds after call is issued and every 2 seconds thereafter.



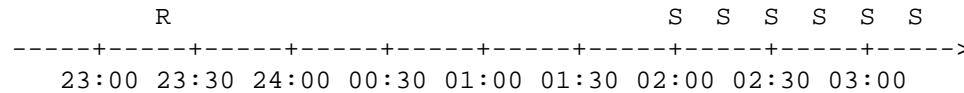
[Time (in Seconds)]

To achieve this type of scheduling using the specified facilities, one would allocate a per-process timer based on clock ID `CLOCK_REALTIME`. Then the timer would be armed via a call to `timer_settime()` with `TIMER_ABSTIME` flag reset, and with an initial expiration value of 15 seconds and a repetition interval of 2 seconds.

#### — Periodic Timer (Absolute Time): Work Shift-related Processing

Resource utilization data is useful when time to perform experiments is being scheduled at a facility. With a periodic timer with an absolute initial expiration time, a module can be scheduled at the beginning of a work shift to gather resource utilization data throughout the shift. This data can be used to allocate resources effectively to minimize bottlenecks and delays and maximize facility throughput.

Program schedule requested for 2:00 a.m. and every 15 minutes thereafter.



[Time of Day]

To achieve this type of scheduling using the specified facilities, one would allocate a per-process timer based on clock ID `CLOCK_REALTIME`. Then the timer would be armed via a call to `timer_settime()` with `TIMER_ABSTIME` flag set, and with an initial expiration value equal to 2:00 a.m. and a repetition interval equal to 15 minutes.

#### • Relationship of Timers to Clocks

The relationship between clocks and timers armed with an absolute time is straightforward: a timer expiration signal is requested when the associated clock reaches or exceeds the specified time. The relationship between clocks and timers armed with a relative time (an interval) is less obvious, but not unintuitive. In this case, a timer expiration signal is requested when the specified interval, *as measured by the associated clock*, has passed. For the required `CLOCK_REALTIME` clock, this allows timer expiration signals to be requested at specified “wall clock” times (absolute), or when a specified interval of “realtime” has passed (relative). For an implementation-defined clock—say, a process virtual time clock—timer expirations could be requested when the process has used a specified total amount of virtual time (absolute), or when it has used a specified *additional* amount of virtual time (relative).

The interfaces also allow flexibility in the implementation of the functions. For example, an implementation could convert all absolute times to intervals by subtracting the clock value at the time of the call from the requested expiration time and “counting down” at the supported resolution. Or it could convert all relative times to absolute expiration time by adding in the clock value at the time of the call and comparing the clock value to the expiration time at the supported resolution. Or it might even choose to maintain absolute times as absolute and compare them to the clock value at the supported resolution for absolute timers, and maintain relative times as intervals and count them down at the

5747 resolution supported for relative timers. The choice will be driven by efficiency  
5748 considerations and the underlying hardware or software clock implementation.

5749 • Data Definitions for Clocks and Timers

5750 IEEE Std 1003.1-2001 uses a time representation capable of supporting nanosecond resolution  
5751 timers for the following reasons:

5752 — To enable IEEE Std 1003.1-2001 to represent those computer systems already using  
5753 nanosecond or submicrosecond resolution clocks.

5754 — To accommodate those per-process timers that might need nanoseconds to specify an  
5755 absolute value of system-wide clocks, even though the resolution of the per-process timer  
5756 may only be milliseconds, or *vice versa*.

5757 — Because the number of nanoseconds in a second can be represented in 32 bits.

5758 Time values are represented in the **timespec** structure. The *tv\_sec* member is of type **time\_t**  
5759 so that this member is compatible with time values used by POSIX.1 functions and the ISO C  
5760 standard. The *tv\_nsec* member is a **signed long** in order to simplify and clarify code that  
5761 decrements or finds differences of time values. Note that because 1 billion (number of  
5762 nanoseconds per second) is less than half of the value representable by a signed 32-bit value,  
5763 it is always possible to add two valid fractional seconds represented as integral nanoseconds  
5764 without overflowing the signed 32-bit value.

5765 A maximum allowable resolution for the CLOCK\_REALTIME clock of 20 ms (1/50 seconds)  
5766 was chosen to allow line frequency clocks in European countries to be conforming. 60 Hz  
5767 clocks in the U.S. will also be conforming, as will finer granularity clocks, although a Strictly  
5768 Conforming Application cannot assume a granularity of less than 20 ms (1/50 seconds).

5769 The minimum allowable maximum time allowed for the CLOCK\_REALTIME clock and the  
5770 function *nanosleep()*, and timers created with *clock\_id*=CLOCK\_REALTIME, is determined by  
5771 the fact that the *tv\_sec* member is of type **time\_t**.

5772 IEEE Std 1003.1-2001 specifies that timer expirations must not be delivered early, and  
5773 *nanosleep()* must not return early due to quantization error. IEEE Std 1003.1-2001 discusses  
5774 the various implementations of *alarm()* in the rationale and states that implementations that  
5775 do not allow alarm signals to occur early are the most appropriate, but refrained from  
5776 mandating this behavior. Because of the importance of predictability to realtime applications,  
5777 IEEE Std 1003.1-2001 takes a stronger stance.

5778 The developers of IEEE Std 1003.1-2001 considered using a time representation that differs  
5779 from POSIX.1b in the second 32 bit of the 64-bit value. Whereas POSIX.1b defines this field  
5780 as a fractional second in nanoseconds, the other methodology defines this as a binary fraction  
5781 of one second, with the radix point assumed before the most significant bit.

5782 POSIX.1b is a software, source-level standard and most of the benefits of the alternate  
5783 representation are enjoyed by hardware implementations of clocks and algorithms. It was  
5784 felt that mandating this format for POSIX.1b clocks and timers would unnecessarily burden  
5785 the application writer with writing, possibly non-portable, multiple precision arithmetic  
5786 packages to perform conversion between binary fractions and integral units such as  
5787 nanoseconds, milliseconds, and so on.

**Rationale for the Monotonic Clock**

For those applications that use time services to achieve realtime behavior, changing the value of the clock on which these services rely may cause erroneous timing behavior. For these applications, it is necessary to have a monotonic clock which cannot run backwards, and which has a maximum clock jump that is required to be documented by the implementation. Additionally, it is desirable (but not required by IEEE Std 1003.1-2001) that the monotonic clock increases its value uniformly. This clock should not be affected by changes to the system time; for example, to synchronize the clock with an external source or to account for leap seconds. Such changes would cause errors in the measurement of time intervals for those time services that use the absolute value of the clock.

One could argue that by defining the behavior of time services when the value of a clock is changed, deterministic realtime behavior can be achieved. For example, one could specify that relative time services should be unaffected by changes in the value of a clock. However, there are time services that are based upon an absolute time, but that are essentially intended as relative time services. For example, *pthread\_cond\_timedwait()* uses an absolute time to allow it to wake up after the required interval despite spurious wakeups. Although sometimes the *pthread\_cond\_timedwait()* timeouts are absolute in nature, there are many occasions in which they are relative, and their absolute value is determined from the current time plus a relative time interval. In this latter case, if the clock changes while the thread is waiting, the wait interval will not be the expected length. If a *pthread\_cond\_timedwait()* function were created that would take a relative time, it would not solve the problem because to retain the intended “deadline” a thread would need to compensate for latency due to the spurious wakeup, and preemption between wakeup and the next wait.

The solution is to create a new monotonic clock, whose value does not change except for the regular ticking of the clock, and use this clock for implementing the various relative timeouts that appear in the different POSIX interfaces, as well as allow *pthread\_cond\_timedwait()* to choose this new clock for its timeout. A new *clock\_nanosleep()* function is created to allow an application to take advantage of this newly defined clock. Notice that the monotonic clock may be implemented using the same hardware clock as the system clock.

Relative timeouts for *sigtimedwait()* and *aio\_suspend()* have been redefined to use the monotonic clock, if present. The *alarm()* function has not been redefined, because the same effect but with better resolution can be achieved by creating a timer (for which the appropriate clock may be chosen).

The *pthread\_cond\_timedwait()* function has been treated in a different way, compared to other functions with absolute timeouts, because it is used to wait for an event, and thus it may have a deadline, while the other timeouts are generally used as an error recovery mechanism, and for them the use of the monotonic clock is not so important. Since the desired timeout for the *pthread\_cond\_timedwait()* function may either be a relative interval or an absolute time of day deadline, a new initialization attribute has been created for condition variables to specify the clock that is used for measuring the timeout in a call to *pthread\_cond\_timedwait()*. In this way, if a relative timeout is desired, the monotonic clock will be used; if an absolute deadline is required instead, the *CLOCK\_REALTIME* or another appropriate clock may be used. This capability has not been added to other functions with absolute timeouts because for those functions the expected use of the timeout is mostly to prevent errors, and not so often to meet precise deadlines. As a consequence, the complexity of adding this capability is not justified by its perceived application usage.

The *nanosleep()* function has not been modified with the introduction of the monotonic clock. Instead, a new *clock\_nanosleep()* function has been created, in which the desired clock may be specified in the function call.



- History of Resolution Issues

Due to the shift from relative to absolute timeouts in IEEE Std 1003.1d-1999, the amendments to the `sem_timedwait()`, `pthread_mutex_timedlock()`, `mq_timedreceive()`, and `mq_timedsend()` functions of that standard have been removed. Those amendments specified that `CLOCK_MONOTONIC` would be used for the (relative) timeouts if the Monotonic Clock option was supported.

Having these functions continue to be tied solely to `CLOCK_MONOTONIC` would not work. Since the absolute value of a time value obtained from `CLOCK_MONOTONIC` is unspecified, under the absolute timeouts interface, applications would behave differently depending on whether the Monotonic Clock option was supported or not (because the absolute value of the clock would have different meanings in either case).

Two options were considered:

1. Leave the current behavior unchanged, which specifies the `CLOCK_REALTIME` clock for these (absolute) timeouts, to allow portability of applications between implementations supporting or not the Monotonic Clock option.
2. Modify these functions in the way that `pthread_cond_timedwait()` was modified to allow a choice of clock, so that an application could use `CLOCK_REALTIME` when it is trying to achieve an absolute timeout and `CLOCK_MONOTONIC` when it is trying to achieve a relative timeout.

It was decided that the features of `CLOCK_MONOTONIC` are not as critical to these functions as they are to `pthread_cond_timedwait()`. The `pthread_cond_timedwait()` function is given a relative timeout; the timeout may represent a deadline for an event. When these functions are given relative timeouts, the timeouts are typically for error recovery purposes and need not be so precise.

Therefore, it was decided that these functions should be tied to `CLOCK_REALTIME` and not complicated by being given a choice of clock.

## Execution Time Monitoring

- Introduction

The main goals of the execution time monitoring facilities defined in this chapter are to measure the execution time of processes and threads and to allow an application to establish CPU time limits for these entities.

The analysis phase of time-critical realtime systems often relies on the measurement of execution times of individual threads or processes to determine whether the timing requirements will be met. Also, performance analysis techniques for soft deadline realtime systems rely heavily on the determination of these execution times. The execution time monitoring functions provide application developers with the ability to measure these execution times online and open the possibility of dynamic execution-time analysis and system reconfiguration, if required.

The second goal of allowing an application to establish execution time limits for individual processes or threads and detecting when they overrun allows program robustness to be increased by enabling online checking of the execution times.

If errors are detected—possibly because of erroneous program constructs, the existence of errors in the analysis phase, or a burst of event arrivals—online detection and recovery is possible in a portable way. This feature can be extremely important for many time-critical applications. Other applications require trapping CPU-time errors as a normal way to exit an

algorithm; for instance, some realtime artificial intelligence applications trigger a number of independent inference processes of varying accuracy and speed, limit how long they can run, and pick the best answer available when time runs out. In many periodic systems, overrun processes are simply restarted in the next resource period, after necessary end-of-period actions have been taken. This allows algorithms that are inherently data-dependent to be made predictable.

The interface that appears in this chapter defines a new type of clock, the CPU-time clock, which measures execution time. Each process or thread can invoke the clock and timer functions defined in POSIX.1 to use them. Functions are also provided to access the CPU-time clock of other processes or threads to enable remote monitoring of these clocks. Monitoring of threads of other processes is not supported, since these threads are not visible from outside of their own process with the interfaces defined in POSIX.1.

#### • Execution Time Monitoring Interface

The clock and timer interface defined in POSIX.1 historically only defined one clock, which measures wall-clock time. The requirements for measuring execution time of processes and threads, and setting limits to their execution time by detecting when they overrun, can be accomplished with that interface if a new kind of clock is defined. These new clocks measure execution time, and one is associated with each process and with each thread. The clock functions currently defined in POSIX.1 can be used to read and set these CPU-time clocks, and timers can be created using these clocks as their timing base. These timers can then be used to send a signal when some specified execution time has been exceeded. The CPU-time clocks of each process or thread can be accessed by using the symbols `CLOCK_PROCESS_CPUTIME_ID` or `CLOCK_THREAD_CPUTIME_ID`.

The clock and timer interface defined in POSIX.1 and extended with the new kind of CPU-time clock would only allow processes or threads to access their own CPU-time clocks. However, many realtime systems require the possibility of monitoring the execution time of processes or threads from independent monitoring entities. In order to allow applications to construct independent monitoring entities that do not require cooperation from or modification of the monitored entities, two functions have been added: `clock_getcpuclockid()`, for accessing CPU-time clocks of other processes, and `pthread_getcpuclockid()`, for accessing CPU-time clocks of other threads. These functions return the clock identifier associated with the process or thread specified in the call. These clock IDs can then be used in the rest of the clock function calls.

The clocks accessed through these functions could also be used as a timing base for the creation of timers, thereby allowing independent monitoring entities to limit the CPU time consumed by other entities. However, this possibility would imply additional complexity and overhead because of the need to maintain a timer queue for each process or thread, to store the different expiration times associated with timers created by different processes or threads. The working group decided this additional overhead was not justified by application requirements. Therefore, creation of timers attached to the CPU-time clocks of other processes or threads has been specified as implementation-defined.

#### • Overhead Considerations

The measurement of execution time may introduce additional overhead in the thread scheduling, because of the need to keep track of the time consumed by each of these entities. In library-level implementations of threads, the efficiency of scheduling could be somehow compromised because of the need to make a kernel call, at each context switch, to read the process CPU-time clock. Consequently, a thread creation attribute called *cpu-clock-requirement* was defined, to allow threads to disconnect their respective CPU-time clocks. However, the Ballot Group considered that this attribute itself introduced some overhead,

and that in current implementations it was not worth the effort. Therefore, the attribute was deleted, and thus thread CPU-time clocks are required for all threads if the Thread CPU-Time Clocks option is supported.

- Accuracy of CPU-Time Clocks

The mechanism used to measure the execution time of processes and threads is specified in IEEE Std 1003.1-2001 as implementation-defined. The reason for this is that both the underlying hardware and the implementation architecture have a very strong influence on the accuracy achievable for measuring CPU time. For some implementations, the specification of strict accuracy requirements would represent very large overheads, or even the impossibility of being implemented.

Since the mechanism for measuring execution time is implementation-defined, realtime applications will be able to take advantage of accurate implementations using a portable interface. Of course, strictly conforming applications cannot rely on any particular degree of accuracy, in the same way as they cannot rely on a very accurate measurement of wall clock time. There will always exist applications whose accuracy or efficiency requirements on the implementation are more rigid than the values defined in IEEE Std 1003.1-2001 or any other standard.

In any case, there is a minimum set of characteristics that realtime applications would expect from most implementations. One such characteristic is that the sum of all the execution times of all the threads in a process equals the process execution time, when no CPU-time clocks are disabled. This need not always be the case because implementations may differ in how they account for time during context switches. Another characteristic is that the sum of the execution times of all processes in a system equals the number of processors, multiplied by the elapsed time, assuming that no processor is idle during that elapsed time. However, in some implementations it might not be possible to relate CPU time to elapsed time. For example, in a heterogeneous multi-processor system in which each processor runs at a different speed, an implementation may choose to define each “second” of CPU time to be a certain number of “cycles” that a CPU has executed.

- Existing Practice

Measuring and limiting the execution time of each concurrent activity are common features of most industrial implementations of realtime systems. Almost all critical realtime systems are currently built upon a cyclic executive. With this approach, a regular timer interrupt kicks off the next sequence of computations. It also checks that the current sequence has completed. If it has not, then some error recovery action can be undertaken (or at least an overrun is avoided). Current software engineering principles and the increasing complexity of software are driving application developers to implement these systems on multi-threaded or multi-process operating systems. Therefore, if a POSIX operating system is to be used for this type of application, then it must offer the same level of protection.

Execution time clocks are also common in most UNIX implementations, although these clocks usually have requirements different from those of realtime applications. The POSIX.1 *times()* function supports the measurement of the execution time of the calling process, and its terminated child processes. This execution time is measured in clock ticks and is supplied as two different values with the user and system execution times, respectively. BSD supports the function *getrusage()*, which allows the calling process to get information about the resources used by itself and/or all of its terminated child processes. The resource usage includes user and system CPU time. Some UNIX systems have options to specify high resolution (up to one microsecond) CPU-time clocks using the *times()* or the *getrusage()* functions.

The *times()* and *getrusage()* interfaces do not meet important realtime requirements, such as the possibility of monitoring execution time from a different process or thread, or the possibility of detecting an execution time overrun. The latter requirement is supported in some UNIX implementations that are able to send a signal when the execution time of a process has exceeded some specified value. For example, BSD defines the functions *getitimer()* and *setitimer()*, which can operate either on a realtime clock (wall-clock), or on virtual-time or profile-time clocks which measure CPU time in two different ways. These functions do not support access to the execution time of other processes.

IBM's MVS operating system supports per-process and per-thread execution time clocks. It also supports limiting the execution time of a given process.

Given all this existing practice, the working group considered that the POSIX.1 clocks and timers interface was appropriate to meet most of the requirements that realtime applications have for execution time clocks. Functions were added to get the CPU time clock IDs, and to allow/disallow the thread CPU-time clocks (in order to preserve the efficiency of some implementations of threads).

#### • Clock Constants

The definition of the manifest constants *CLOCK\_PROCESS\_CPUTIME\_ID* and *CLOCK\_THREAD\_CPUTIME\_ID* allows processes or threads, respectively, to access their own execution-time clocks. However, given a process or thread, access to its own execution-time clock is also possible if the clock ID of this clock is obtained through a call to *clock\_getcpuclockid()* or *pthread\_getcpuclockid()*. Therefore, these constants are not necessary and could be deleted to make the interface simpler. Their existence saves one system call in the first access to the CPU-time clock of each process or thread. The working group considered this issue and decided to leave the constants in IEEE Std 1003.1-2001 because they are closer to the POSIX.1b use of clock identifiers.

#### • Library Implementations of Threads

In library implementations of threads, kernel entities and library threads can coexist. In this case, if the CPU-time clocks are supported, most of the clock and timer functions will need to have two implementations: one in the thread library, and one in the system calls library. The main difference between these two implementations is that the thread library implementation will have to deal with clocks and timers that reside in the thread space, while the kernel implementation will operate on timers and clocks that reside in kernel space. In the library implementation, if the clock ID refers to a clock that resides in the kernel, a kernel call will have to be made. The correct version of the function can be chosen by specifying the appropriate order for the libraries during the link process.

#### • History of Resolution Issues: Deletion of the *enable* Attribute

In early proposals, consideration was given to inclusion of an attribute called *enable* for CPU-time clocks. This would allow implementations to avoid the overhead of measuring execution time for those processes or threads for which this measurement was not required. However, this is unnecessary since processes are already required to measure execution time by the POSIX.1 *times()* function. Consequently, the *enable* attribute is not present.

**Rationale Relating to Timeouts**

- Requirements for Timeouts

Realtime systems which must operate reliably over extended periods without human intervention are characteristic in embedded applications such as avionics, machine control, and space exploration, as well as more mundane applications such as cable TV, security systems, and plant automation. A multi-tasking paradigm, in which many independent and/or cooperating software functions relinquish the processor(s) while waiting for a specific stimulus, resource, condition, or operation completion, is very useful in producing well engineered programs for such systems. For such systems to be robust and fault-tolerant, expected occurrences that are unduly delayed or that never occur must be detected so that appropriate recovery actions may be taken. This is difficult if there is no way for a task to regain control of a processor once it has relinquished control (blocked) awaiting an occurrence which, perhaps because of corrupted code, hardware malfunction, or latent software bugs, will not happen when expected. Therefore, the common practice in realtime operating systems is to provide a capability to time out such blocking services. Although there are several methods to achieve this already defined by POSIX, none are as reliable or efficient as initiating a timeout simultaneously with initiating a blocking service. This is especially critical in hard-realtime embedded systems because the processors typically have little time reserve, and allowed fault recovery times are measured in milliseconds rather than seconds.

The working group largely agreed that such timeouts were necessary and ought to become part of IEEE Std 1003.1-2001, particularly vendors of realtime operating systems whose customers had already expressed a strong need for timeouts. There was some resistance to inclusion of timeouts in IEEE Std 1003.1-2001 because the desired effect, fault tolerance, could, in theory, be achieved using existing facilities and alternative software designs, but there was no compelling evidence that realtime system designers would embrace such designs at the sacrifice of performance and/or simplicity.

- Which Services should be Timed Out?

Originally, the working group considered the prospect of providing timeouts on all blocking services, including those currently existing in POSIX.1, POSIX.1b, and POSIX.1c, and future interfaces to be defined by other working groups, as sort of a general policy. This was rather quickly rejected because of the scope of such a change, and the fact that many of those services would not normally be used in a realtime context. More traditional timesharing solutions to timeout would suffice for most of the POSIX.1 interfaces, while others had asynchronous alternatives which, while more complex to utilize, would be adequate for some realtime and all non-realtime applications.

The list of potential candidates for timeouts was narrowed to the following for further consideration:

- POSIX.1b

- *sem\_wait()*

- *mq\_receive()*

- *mq\_send()*

- *lio\_listio()*

- *aio\_suspend()*

- *sigwait()* (timeout already implemented by *sigtimedwait()*)

6065 — POSIX.1c

6066 — *pthread\_mutex\_lock()*

6067 — *pthread\_join()*

6068 — *pthread\_cond\_wait()* (timeout already implemented by *pthread\_cond\_timedwait()*)

6069 — POSIX.1

6070 — *read()*

6071 — *write()*

6072 After further review by the working group, the *lio\_listio()*, *read()*, and *write()* functions (all

6073 forms of blocking synchronous I/O) were eliminated from the list because of the following:

6074 — Asynchronous alternatives exist

6075 — Timeouts can be implemented, albeit non-portably, in device drivers

6076 — A strong desire not to introduce modifications to POSIX.1 interfaces

6077 The working group ultimately rejected *pthread\_join()* since both that interface and a timed

6078 variant of that interface are non-minimal and may be implemented as a function. See below

6079 for a library implementation of *pthread\_join()*.

6080 Thus, there was a consensus among the working group members to add timeouts to 4 of the

6081 remaining 5 functions (the timeout for *aio\_suspend()* was ultimately added directly to

6082 POSIX.1b, while the others were added by POSIX.1d). However, *pthread\_mutex\_lock()*

6083 remained contentious.

6084 Many feel that *pthread\_mutex\_lock()* falls into the same class as the other functions; that is, it

6085 is desirable to time out a mutex lock because a mutex may fail to be unlocked due to errant or

6086 corrupted code in a critical section (looping or branching outside of the unlock code), and

6087 therefore is equally in need of a reliable, simple, and efficient timeout. In fact, since mutexes

6088 are intended to guard small critical sections, most *pthread\_mutex\_lock()* calls would be

6089 expected to obtain the lock without blocking nor utilizing any kernel service, even in

6090 implementations of threads with global contention scope; the timeout alternative need only

6091 be considered after it is determined that the thread must block.

6092 Those opposed to timing out mutexes feel that the very simplicity of the mutex is

6093 compromised by adding a timeout semantic, and that to do so is senseless. They claim that if

6094 a timed mutex is really deemed useful by a particular application, then it can be constructed

6095 from the facilities already in POSIX.1b and POSIX.1c. The following two C-language library

6096 implementations of mutex locking with timeout represent the solutions offered (in both

6097 implementations, the timeout parameter is specified as absolute time, not relative time as in

6098 the proposed POSIX.1c interfaces).

### • Spinlock Implementation

```

6099
6100     #include <pthread.h>
6101     #include <time.h>
6102     #include <errno.h>
6103
6104     int pthread_mutex_timedlock(pthread_mutex_t *mutex,
6105                                const struct timespec *timeout)
6106     {
6107         struct timespec timenow;
6108
6109         while (pthread_mutex_trylock(mutex) == EBUSY)
6110         {
6111             clock_gettime(CLOCK_REALTIME, &timenow);
6112             if (timespec_cmp(&timenow, timeout) >= 0)
6113             {
6114                 return ETIMEDOUT;
6115             }
6116             pthread_yield();
6117         }
6118         return 0;
6119     }

```

The Spinlock implementation is generally unsuitable for any application using priority-based thread scheduling policies such as SCHED\_FIFO or SCHED\_RR, since the mutex could currently be held by a thread of lower priority within the same allocation domain, but since the waiting thread never blocks, only threads of equal or higher priority will ever run, and the mutex cannot be unlocked. Setting priority inheritance or priority ceiling protocol on the mutex does not solve this problem, since the priority of a mutex owning thread is only boosted if higher priority threads are blocked waiting for the mutex; clearly not the case for this spinlock.

### • Condition Wait Implementation

```

6127     #include <pthread.h>
6128     #include <time.h>
6129     #include <errno.h>
6130
6131     struct timed_mutex
6132     {
6133         int locked;
6134         pthread_mutex_t mutex;
6135         pthread_cond_t cond;
6136     };
6137
6138     typedef struct timed_mutex timed_mutex_t;
6139
6140     int timed_mutex_lock(timed_mutex_t *tm,
6141                        const struct timespec *timeout)
6142     {
6143         int timedout=FALSE;
6144         int error_status;
6145
6146         pthread_mutex_lock(&tm->mutex);
6147
6148         while (tm->locked && !timedout)
6149         {
6150             if ((error_status=pthread_cond_timedwait(&tm->cond,

```

```

6146         &tm->mutex,
6147         timeout))!=0)
6148     {
6149         if (error_status==ETIMEDOUT) timedout = TRUE;
6150     }
6151 }
6152
6153 if(timedout)
6154 {
6155     pthread_mutex_unlock(&tm->mutex);
6156     return ETIMEDOUT;
6157 }
6158 else
6159 {
6160     tm->locked = TRUE;
6161     pthread_mutex_unlock(&tm->mutex);
6162     return 0;
6163 }
6164
6165 void timed_mutex_unlock(timed_mutex_t *tm)
6166 {
6167     pthread_mutex_lock(&tm->mutex); / for case assignment not atomic /
6168     tm->locked = FALSE;
6169     pthread_mutex_unlock(&tm->mutex);
6170     pthread_cond_signal(&tm->cond);
6171 }

```

The Condition Wait implementation effectively substitutes the *pthread\_cond\_timedwait()* function (which is currently timed out) for the desired *pthread\_mutex\_timedlock()*. Since waits on condition variables currently do not include protocols which avoid priority inversion, this method is generally unsuitable for realtime applications because it does not provide the same priority inversion protection as the untimed *pthread\_mutex\_lock()*. Also, for any given implementations of the current mutex and condition variable primitives, this library implementation has a performance cost at least 2.5 times that of the untimed *pthread\_mutex\_lock()* even in the case where the timed mutex is readily locked without blocking (the interfaces required for this case are shown in bold). Even in uniprocessors or where assignment is atomic, at least an additional *pthread\_cond\_signal()* is required. *pthread\_mutex\_timedlock()* could be implemented at effectively no performance penalty in this case because the timeout parameters need only be considered after it is determined that the mutex cannot be locked immediately.

Thus it has not yet been shown that the full semantics of mutex locking with timeout can be efficiently and reliably achieved using existing interfaces. Even if the existence of an acceptable library implementation were proven, it is difficult to justify why the interface itself should not be made portable, especially considering approval for the other four timeouts.



- Rationale for Library Implementation of *pthread\_timedjoin()*

Library implementation of *pthread\_timedjoin()*:

```

6189
6190
6191     /*
6192     * Construct a thread variety entirely from existing functions
6193     * with which a join can be done, allowing the join to time out.
6194     */
6195     #include <pthread.h>
6196     #include <time.h>
6197
6198     struct timed_thread {
6199         pthread_t t;
6200         pthread_mutex_t m;
6201         int exiting;
6202         pthread_cond_t exit_c;
6203         void *(*start_routine)(void *arg);
6204         void *arg;
6205         void *status;
6206     };
6207
6208     typedef struct timed_thread *timed_thread_t;
6209     static pthread_key_t timed_thread_key;
6210     static pthread_once_t timed_thread_once = PTHREAD_ONCE_INIT;
6211
6212     static void timed_thread_init()
6213     {
6214         pthread_key_create(&timed_thread_key, NULL);
6215     }
6216
6217     static void *timed_thread_start_routine(void *args)
6218     /*
6219     * Routine to establish thread-specific data value and run the actual
6220     * thread start routine which was supplied to timed_thread_create().
6221     */
6222     {
6223         timed_thread_t tt = (timed_thread_t) args;
6224
6225         pthread_once(&timed_thread_once, timed_thread_init);
6226         pthread_setspecific(timed_thread_key, (void *)tt);
6227         timed_thread_exit((tt->start_routine)(tt->arg));
6228     }
6229
6230     int timed_thread_create(timed_thread_t ttp, const pthread_attr_t *attr,
6231         void *(*start_routine)(void *), void *arg)
6232
6233     /*
6234     * Allocate a thread which can be used with timed_thread_join().
6235     */
6236     {
6237         timed_thread_t tt;
6238         int result;
6239
6240         tt = (timed_thread_t) malloc(sizeof(struct timed_thread));
6241         pthread_mutex_init(&tt->m, NULL);
6242         tt->exiting = FALSE;
6243         pthread_cond_init(&tt->exit_c, NULL);

```

```

6236         tt->start_routine = start_routine;
6237         tt->arg = arg;
6238         tt->status = NULL;

6239         if ((result = pthread_create(&tt->t, attr,
6240             timed_thread_start_routine, (void *)tt)) != 0) {
6241             free(tt);
6242             return result;
6243         }

6244         pthread_detach(tt->t);
6245         ttp = tt;
6246         return 0;
6247     }

6248     int timed_thread_join(timed_thread_t tt,
6249         struct timespec *timeout,
6250         void **status)
6251     {
6252         int result;

6253         pthread_mutex_lock(&tt->m);
6254         result = 0;
6255         /*
6256          * Wait until the thread announces that it is exiting,
6257          * or until timeout.
6258          */
6259         while (result == 0 && ! tt->exiting) {
6260             result = pthread_cond_timedwait(&tt->exit_c, &tt->m, timeout);
6261         }
6262         pthread_mutex_unlock(&tt->m);
6263         if (result == 0 && tt->exiting) {
6264             *status = tt->status;
6265             free((void *)tt);
6266             return result;
6267         }
6268         return result;
6269     }

6270     void timed_thread_exit(void *status)
6271     {
6272         timed_thread_t tt;
6273         void *specific;

6274         if ((specific=pthread_getspecific(timed_thread_key)) == NULL){
6275             /*
6276              * Handle cases which won't happen with correct usage.
6277              */
6278             pthread_exit( NULL);
6279         }
6280         tt = (timed_thread_t) specific;
6281         pthread_mutex_lock(&tt->m);
6282         /*
6283          * Tell a joiner that we're exiting.
6284          */

```

```

6285         tt->status = status;
6286         tt->exiting = TRUE;
6287         pthread_cond_signal(&tt->exit_c);
6288         pthread_mutex_unlock(&tt->m);
6289         /*
6290          * Call pthread_exit() to call destructors and really
6291          * exit the thread.
6292          */
6293         pthread_exit(NULL);
6294     }

```

6295 The *pthread\_join()* C-language example shown above demonstrates that it is possible, using  
 6296 existing pthread facilities, to construct a variety of thread which allows for joining such a  
 6297 thread, but which allows the join operation to time out. It does this by using a  
 6298 *pthread\_cond\_timedwait()* to wait for the thread to exit. A **timed\_thread\_t** descriptor structure  
 6299 is used to pass parameters from the creating thread to the created thread, and from the  
 6300 exiting thread to the joining thread. This implementation is roughly equivalent to what a  
 6301 normal *pthread\_join()* implementation would do, with the single change being that  
 6302 *pthread\_cond\_timedwait()* is used in place of a simple *pthread\_cond\_wait()*.

6303 Since it is possible to implement such a facility entirely from existing pthread interfaces, and  
 6304 with roughly equal efficiency and complexity to an implementation which would be  
 6305 provided directly by a pthreads implementation, it was the consensus of the working group  
 6306 members that any *pthread\_timedjoin()* facility would be unnecessary, and should not be  
 6307 provided.

#### 6308 • Form of the Timeout Interfaces

6309 The working group considered a number of alternative ways to add timeouts to blocking  
 6310 services. At first, a system interface which would specify a one-shot or persistent timeout to  
 6311 be applied to subsequent blocking services invoked by the calling process or thread was  
 6312 considered because it allowed all blocking services to be timed out in a uniform manner with  
 6313 a single additional interface; this was rather quickly rejected because it could easily result in  
 6314 the wrong services being timed out.

6315 It was suggested that a timeout value might be specified as an attribute of the object  
 6316 (semaphore, mutex, message queue, and so on), but there was no consensus on this, either on  
 6317 a case-by-case basis or for all timeouts.

6318 Looking at the two existing timeouts for blocking services indicates that the working group  
 6319 members favor a separate interface for the timed version of a function. However,  
 6320 *pthread\_cond\_timedwait()* utilizes an absolute timeout value while *sigtimedwait()* uses a  
 6321 relative timeout value. The working group members agreed that relative timeout values are  
 6322 appropriate where the timeout mechanism's primary use was to deal with an unexpected or  
 6323 error situation, but they are inappropriate when the timeout must expire at a particular time,  
 6324 or before a specific deadline. For the timeouts being introduced in IEEE Std 1003.1-2001, the  
 6325 working group considered allowing both relative and absolute timeouts as is done with  
 6326 POSIX.1b timers, but ultimately favored the simpler absolute timeout form.

6327 An absolute time measure can be easily implemented on top of an interface that specifies  
 6328 relative time, by reading the clock, calculating the difference between the current time and  
 6329 the desired wake-up time, and issuing a relative timeout call. But there is a race condition  
 6330 with this approach because the thread could be preempted after reading the clock, but before  
 6331 making the timed-out call; in this case, the thread would be awakened later than it should  
 6332 and, thus, if the wake-up time represented a deadline, it would miss it.

There is also a race condition when trying to build a relative timeout on top of an interface that specifies absolute timeouts. In this case, the clock would have to be read to calculate the absolute wake-up time as the sum of the current time plus the relative timeout interval. In this case, if the thread is preempted after reading the clock but before making the timed-out call, the thread would be awakened earlier than desired.

But the race condition with the absolute timeouts interface is not as bad as the one that happens with the relative timeout interface, because there are simple workarounds. For the absolute timeouts interface, if the timing requirement is a deadline, the deadline can still be met because the thread woke up earlier than the deadline. If the timeout is just used as an error recovery mechanism, the precision of timing is not really important. If the timing requirement is that between actions A and B a minimum interval of time must elapse, the absolute timeout interface can be safely used by reading the clock after action A has been started. It could be argued that, since the call with the absolute timeout is atomic from the application point of view, it is not possible to read the clock after action A, if this action is part of the timed-out call. But looking at the nature of the calls for which timeouts are specified (locking a mutex, waiting for a semaphore, waiting for a message, or waiting until there is space in a message queue), the timeouts that an application would build on these actions would not be triggered by these actions themselves, but by some other external action. For example, if waiting for a message to arrive to a message queue, and waiting for at least 20 milliseconds, this time interval would start to be counted from some event that would trigger both the action that produces the message, as well as the action that waits for the message to arrive, and not by the wait-for-message operation itself. In this case, the workaround proposed above could be used.

For these reasons, the absolute timeout is preferred over the relative timeout interface.

## **B.2.9 Threads**

Threads will normally be more expensive than subroutines (or functions, routines, and so on) if specialized hardware support is not provided. Nevertheless, threads should be sufficiently efficient to encourage their use as a medium to fine-grained structuring mechanism for parallelism in an application. Structuring an application using threads then allows it to take immediate advantage of any underlying parallelism available in the host environment. This means implementors are encouraged to optimize for fast execution at the possible expense of efficient utilization of storage. For example, a common thread creation technique is to cache appropriate thread data structures. That is, rather than releasing system resources, the implementation retains these resources and reuses them when the program next asks to create a new thread. If this reuse of thread resources is to be possible, there has to be very little unique state associated with each thread, because any such state has to be reset when the thread is reused.

### **Thread Creation Attributes**

Attributes objects are provided for threads, mutexes, and condition variables as a mechanism to support probable future standardization in these areas without requiring that the interface itself be changed.

Attributes objects provide clean isolation of the configurable aspects of threads. For example, “stack size” is an important attribute of a thread, but it cannot be expressed portably. When porting a threaded program, stack sizes often need to be adjusted. The use of attributes objects can help by allowing the changes to be isolated in a single place, rather than being spread across every instance of thread creation.

Attributes objects can be used to set up *classes* of threads with similar attributes; for example, “threads with large stacks and high priority” or “threads with minimal stacks”. These classes can be defined in a single place and then referenced wherever threads need to be created. Changes to “class” decisions become straightforward, and detailed analysis of each *pthread\_create()* call is not required.

The attributes objects are defined as opaque types as an aid to extensibility. If these objects had been specified as structures, adding new attributes would force recompilation of all multi-threaded programs when the attributes objects are extended; this might not be possible if different program components were supplied by different vendors.

Additionally, opaque attributes objects present opportunities for improving performance. Argument validity can be checked once when attributes are set, rather than each time a thread is created. Implementations will often need to cache kernel objects that are expensive to create. Opaque attributes objects provide an efficient mechanism to detect when cached objects become invalid due to attribute changes.

Because assignment is not necessarily defined on a given opaque type, implementation-defined default values cannot be defined in a portable way. The solution to this problem is to allow attribute objects to be initialized dynamically by attributes object initialization functions, so that default values can be supplied automatically by the implementation.

The following proposal was provided as a suggested alternative to the supplied attributes:

1. Maintain the style of passing a parameter formed by the bitwise-inclusive OR of flags to the initialization routines (*pthread\_create()*, *pthread\_mutex\_init()*, *pthread\_cond\_init()*). The parameter containing the flags should be an opaque type for extensibility. If no flags are set in the parameter, then the objects are created with default characteristics. An implementation may specify implementation-defined flag values and associated behavior.
2. If further specialization of mutexes and condition variables is necessary, implementations may specify additional procedures that operate on the **pthread\_mutex\_t** and **pthread\_cond\_t** objects (instead of on attributes objects).

The difficulties with this solution are:

1. A bitmask is not opaque if bits have to be set into bit-vector attributes objects using explicitly-coded bitwise-inclusive OR operations. If the set of options exceeds an **int**, application programmers need to know the location of each bit. If bits are set or read by encapsulation (that is, *get\*()* or *set\*()* functions), then the bitmask is merely an implementation of attributes objects as currently defined and should not be exposed to the programmer.
2. Many attributes are not Boolean or very small integral values. For example, scheduling policy may be placed in 3 bits or 4 bits, but priority requires 5 bits or more, thereby taking up at least 8 bits out of a possible 16 bits on machines with 16-bit integers. Because of this, the bitmask can only reasonably control whether particular attributes are set or not, and it cannot serve as the repository of the value itself. The value needs to be specified as a function parameter (which is non-extensible), or by setting a structure field (which is non-opaque), or by *get\*()* and *set\*()* functions (making the bitmask a redundant addition to the attributes objects).

Stack size is defined as an optional attribute because the very notion of a stack is inherently machine-dependent. Some implementations may not be able to change the size of the stack, for example, and others may not need to because stack pages may be discontinuous and can be allocated and released on demand.

The attribute mechanism has been designed in large measure for extensibility. Future extensions to the attribute mechanism or to any attributes object defined in IEEE Std 1003.1-2001 have to be done with care so as not to affect binary-compatibility.

Attribute objects, even if allocated by means of dynamic allocation functions such as *malloc()*, may have their size fixed at compile time. This means, for example, a *pthread\_create()* in an implementation with extensions to the *pthread\_attr\_t* cannot look beyond the area that the binary application assumes is valid. This suggests that implementations should maintain a size field in the attributes object, as well as possibly version information, if extensions in different directions (possibly by different vendors) are to be accommodated.

### Thread Implementation Models

There are various thread implementation models. At one end of the spectrum is the “library-thread model”. In such a model, the threads of a process are not visible to the operating system kernel, and the threads are not kernel-scheduled entities. The process is the only kernel-scheduled entity. The process is scheduled onto the processor by the kernel according to the scheduling attributes of the process. The threads are scheduled onto the single kernel-scheduled entity (the process) by the runtime library according to the scheduling attributes of the threads. A problem with this model is that it constrains concurrency. Since there is only one kernel-scheduled entity (namely, the process), only one thread per process can execute at a time. If the thread that is executing blocks on I/O, then the whole process blocks.

At the other end of the spectrum is the “kernel-thread model”. In this model, all threads are visible to the operating system kernel. Thus, all threads are kernel-scheduled entities, and all threads can concurrently execute. The threads are scheduled onto processors by the kernel according to the scheduling attributes of the threads. The drawback to this model is that the creation and management of the threads entails operating system calls, as opposed to subroutine calls, which makes kernel threads heavier weight than library threads.

Hybrids of these two models are common. A hybrid model offers the speed of library threads and the concurrency of kernel threads. In hybrid models, a process has some (relatively small) number of kernel scheduled entities associated with it. It also has a potentially much larger number of library threads associated with it. Some library threads may be bound to kernel-scheduled entities, while the other library threads are multiplexed onto the remaining kernel-scheduled entities. There are two levels of thread scheduling:

1. The runtime library manages the scheduling of (unbound) library threads onto kernel-scheduled entities.
2. The kernel manages the scheduling of kernel-scheduled entities onto processors.

For this reason, a hybrid model is referred to as a two-level threads scheduling model. In this model, the process can have multiple concurrently executing threads; specifically, it can have as many concurrently executing threads as it has kernel-scheduled entities.

### Thread-Specific Data

Many applications require that a certain amount of context be maintained on a per-thread basis across procedure calls. A common example is a multi-threaded library routine that allocates resources from a common pool and maintains an active resource list for each thread. The thread-specific data interface provided to meet these needs may be viewed as a two-dimensional array of values with keys serving as the row index and thread IDs as the column index (although the implementation need not work this way).

- Models

Three possible thread-specific data models were considered:

1. No Explicit Support

A standard thread-specific data interface is not strictly necessary to support applications that require per-thread context. One could, for example, provide a hash function that converted a `pthread_t` into an integer value that could then be used to index into a global array of per-thread data pointers. This hash function, in conjunction with `pthread_self()`, would be all the interface required to support a mechanism of this sort. Unfortunately, this technique is cumbersome. It can lead to duplicated code as each set of cooperating modules implements their own per-thread data management schemes.

2. Single (**void \***) Pointer

Another technique would be to provide a single word of per-thread storage and a pair of functions to fetch and store the value of this word. The word could then hold a pointer to a block of per-thread memory. The allocation, partitioning, and general use of this memory would be entirely up to the application. Although this method is not as problematic as technique 1, it suffers from interoperability problems. For example, all modules using the per-thread pointer would have to agree on a common usage protocol.

3. Key/Value Mechanism

This method associates an opaque key (for example, stored in a variable of type `pthread_key_t`) with each per-thread datum. These keys play the role of identifiers for per-thread data. This technique is the most generic and avoids the problems noted above, albeit at the cost of some complexity.

The primary advantage of the third model is its information hiding properties. Modules using this model are free to create and use their own key(s) independent of all other such usage, whereas the other models require that all modules that use thread-specific context explicitly cooperate with all other such modules. The data-independence provided by the third model is worth the additional interface.

- Requirements

It is important that it be possible to implement the thread-specific data interface without the use of thread private memory. To do otherwise would increase the weight of each thread, thereby limiting the range of applications for which the threads interfaces provided by IEEE Std 1003.1-2001 is appropriate.

The values that one binds to the key via `pthread_setspecific()` may, in fact, be pointers to shared storage locations available to all threads. It is only the key/value bindings that are maintained on a per-thread basis, and these can be kept in any portion of the address space that is reserved for use by the calling thread (for example, on the stack). Thus, no per-thread MMU state is required to implement the interface. On the other hand, there is nothing in the interface specification to preclude the use of a per-thread MMU state if it is available (for example, the key values returned by `pthread_key_create()` could be thread private memory addresses).

- Standardization Issues

Thread-specific data is a requirement for a usable thread interface. The binding described in this section provides a portable thread-specific data mechanism for languages that do not directly support a thread-specific storage class. A binding to IEEE Std 1003.1-2001 for a

language that does include such a storage class need not provide this specific interface.

If a language were to include the notion of thread-specific storage, it would be desirable (but *not* required) to provide an implementation of the pthreads thread-specific data interface based on the language feature. For example, assume that a compiler for a C-like language supports a *private* storage class that provides thread-specific storage. Something similar to the following macros might be used to effect a compatible implementation:

```
#define pthread_key_t          private void *
#define pthread_key_create(key) /* no-op */
#define pthread_setspecific(key,value) (key)=(value)
#define pthread_getspecific(key)      (key)
```

**Note:** For the sake of clarity, this example ignores destructor functions. A correct implementation would have to support them.

## Barriers

### • Background

Barriers are typically used in parallel DO/FOR loops to ensure that all threads have reached a particular stage in a parallel computation before allowing any to proceed to the next stage. Highly efficient implementation is possible on machines which support a “Fetch and Add” operation as described in the referenced Almasi and Gottlieb (1989).

The use of return value PTHREAD\_BARRIER\_SERIAL\_THREAD is shown in the following example:

```
if ( (status=pthread_barrier_wait(&barrier)) ==
    PTHREAD_BARRIER_SERIAL_THREAD) {
    ...serial section
}
else if (status != 0) {
    ...error processing
}
status=pthread_barrier_wait(&barrier);
...
```

This behavior allows a serial section of code to be executed by one thread as soon as all threads reach the first barrier. The second barrier prevents the other threads from proceeding until the serial section being executed by the one thread has completed.

Although barriers can be implemented with mutexes and condition variables, the referenced Almasi and Gottlieb (1989) provides ample illustration that such implementations are significantly less efficient than is possible. While the relative efficiency of barriers may well vary by implementation, it is important that they be recognized in the IEEE Std 1003.1-2001 to facilitate applications portability while providing the necessary freedom to implementors.

### • Lack of Timeout Feature

Alternate versions of most blocking routines have been provided to support watchdog timeouts. No alternate interface of this sort has been provided for barrier waits for the following reasons:

- Multiple threads may use different timeout values, some of which may be indefinite. It is not clear which threads should break through the barrier with a timeout error if and when these timeouts expire.



6559                   • The barrier may become unusable once a thread breaks out of a *pthread\_barrier\_wait()*  
 6560                   with a timeout error. There is, in general, no way to guarantee the consistency of a  
 6561                   barrier's internal data structures once a thread has timed out of a *pthread\_barrier\_wait()*.  
 6562                   Even the inclusion of a special barrier reinitialization function would not help much since  
 6563                   it is not clear how this function would affect the behavior of threads that reach the barrier  
 6564                   between the original timeout and the call to the reinitialization function.

## 6565                   **Spin Locks**

### 6566                   • Background

6567                   Spin locks represent an extremely low-level synchronization mechanism suitable primarily  
 6568                   for use on shared memory multi-processors. It is typically an atomically modified Boolean  
 6569                   value that is set to one when the lock is held and to zero when the lock is freed.

6570                   When a caller requests a spin lock that is already held, it typically spins in a loop testing  
 6571                   whether the lock has become available. Such spinning wastes processor cycles so the lock  
 6572                   should only be held for short durations and not across sleep/block operations. Callers should  
 6573                   unlock spin locks before calling sleep operations.

6574                   Spin locks are available on a variety of systems. The functions included in  
 6575                   IEEE Std 1003.1-2001 are an attempt to standardize that existing practice.

### 6576                   • Lack of Timeout Feature

6577                   Alternate versions of most blocking routines have been provided to support watchdog  
 6578                   timeouts. No alternate interface of this sort has been provided for spin locks for the following  
 6579                   reasons:

6580                   • It is impossible to determine appropriate timeout intervals for spin locks in a portable  
 6581                   manner. The amount of time one can expect to spend spin-waiting is inversely  
 6582                   proportional to the degree of parallelism provided by the system.

6583                   It can vary from a few cycles when each competing thread is running on its own  
 6584                   processor, to an indefinite amount of time when all threads are multiplexed on a single  
 6585                   processor (which is why spin locking is not advisable on uniprocessors).

6586                   • When used properly, the amount of time the calling thread spends waiting on a spin lock  
 6587                   should be considerably less than the time required to set up a corresponding watchdog  
 6588                   timer. Since the primary purpose of spin locks is to provide a low-overhead  
 6589                   synchronization mechanism for multi-processors, the overhead of a timeout mechanism  
 6590                   was deemed unacceptable.

6591                   It was also suggested that an additional *count* argument be provided (on the  
 6592                   *pthread\_spin\_lock()* call) in lieu of a true timeout so that a spin lock call could fail gracefully if  
 6593                   it was unable to apply the lock after *count* attempts. This idea was rejected because it is not  
 6594                   existing practice. Furthermore, the same effect can be obtained with *pthread\_spin\_trylock()*,  
 6595                   as illustrated below:

```

6596         int n = MAX_SPIN;
6597         while ( --n >= 0 )
6598         {
6599             if ( !pthread_spin_try_lock(...) )
6600                 break;
6601         }
6602         if ( n >= 0 )
6603         {
6604             /* Successfully acquired the lock */
6605         }
6606         else
6607         {
6608             /* Unable to acquire the lock */
6609         }

```

- *process-shared* Attribute

The initialization functions associated with most POSIX synchronization objects (for example, mutexes, barriers, and read-write locks) take an attributes object with a *process-shared* attribute that specifies whether or not the object is to be shared across processes. In the draft corresponding to the first balloting round, two separate initialization functions are provided for spin locks, however: one for spin locks that were to be shared across processes (*spin\_init()*), and one for locks that were only used by multiple threads within a single process (*pthread\_spin\_init()*). This was done so as to keep the overhead associated with spin waiting to an absolute minimum. However, the balloting group requested that, since the overhead associated to a bit check was small, spin locks should be consistent with the rest of the synchronization primitives, and thus the *process-shared* attribute was introduced for spin locks.

- Spin Locks versus Mutexes

It has been suggested that mutexes are an adequate synchronization mechanism and spin locks are not necessary. Locking mechanisms typically must trade off the processor resources consumed while setting up to block the thread and the processor resources consumed by the thread while it is blocked. Spin locks require very little resources to set up the blocking of a thread. Existing practice is to simply loop, repeating the atomic locking operation until the lock is available. While the resources consumed to set up blocking of the thread are low, the thread continues to consume processor resources while it is waiting.

On the other hand, mutexes may be implemented such that the processor resources consumed to block the thread are large relative to a spin lock. After detecting that the mutex lock is not available, the thread must alter its scheduling state, add itself to a set of waiting threads, and, when the lock becomes available again, undo all of this before taking over ownership of the mutex. However, while a thread is blocked by a mutex, no processor resources are consumed.

Therefore, spin locks and mutexes may be implemented to have different characteristics. Spin locks may have lower overall overhead for very short-term blocking, and mutexes may have lower overall overhead when a thread will be blocked for longer periods of time. The presence of both interfaces allows implementations with these two different characteristics, both of which may be useful to a particular application.

It has also been suggested that applications can build their own spin locks from the *pthread\_mutex\_trylock()* function:

```
6643         while (pthread_mutex_trylock(&mutex));
```

6644 The apparent simplicity of this construct is somewhat deceiving, however. While the actual  
6645 wait is quite efficient, various guarantees on the integrity of mutex objects (for example,  
6646 priority inheritance rules) may add overhead to the successful path of the trylock operation  
6647 that is not required of spin locks. One could, of course, add an attribute to the mutex to  
6648 bypass such overhead, but the very act of finding and testing this attribute represents more  
6649 overhead than is found in the typical spin lock.

6650 The need to hold spin lock overhead to an absolute minimum also makes it impossible to  
6651 provide guarantees against starvation similar to those provided for mutexes or read-write  
6652 locks. The overhead required to implement such guarantees (for example, disabling  
6653 preemption before spinning) may well exceed the overhead of the spin wait itself by many  
6654 orders of magnitude. If a “safe” spin wait seems desirable, it can always be provided (albeit  
6655 at some performance cost) via appropriate mutex attributes.

## 6656 XSI Supported Functions

6657 On XSI-conformant systems, the following symbolic constants are always defined:

```
6658     _POSIX_READER_WRITER_LOCKS
6659     _POSIX_THREAD_ATTR_STACKADDR
6660     _POSIX_THREAD_ATTR_STACKSIZE
6661     _POSIX_THREAD_PROCESS_SHARED
6662     _POSIX_THREADS
```

6663 Therefore, the following threads functions are always supported:

6664 <i>pthread_atfork()</i>	<i>pthread_key_create()</i>
6665 <i>pthread_attr_destroy()</i>	<i>pthread_key_delete()</i>
6666 <i>pthread_attr_getdetachstate()</i>	<i>pthread_kill()</i>
6667 <i>pthread_attr_getguardsize()</i>	<i>pthread_mutex_destroy()</i>
6668 <i>pthread_attr_getschedparam()</i>	<i>pthread_mutex_init()</i>
6669 <i>pthread_attr_getstack()</i>	<i>pthread_mutex_lock()</i>
6670 <i>pthread_attr_getstackaddr()</i>	<i>pthread_mutex_trylock()</i>
6671 <i>pthread_attr_getstacksize()</i>	<i>pthread_mutex_unlock()</i>
6672 <i>pthread_attr_init()</i>	<i>pthread_mutexattr_destroy()</i>
6673 <i>pthread_attr_setdetachstate()</i>	<i>pthread_mutexattr_getpshared()</i>
6674 <i>pthread_attr_setguardsize()</i>	<i>pthread_mutexattr_gettype()</i>
6675 <i>pthread_attr_setschedparam()</i>	<i>pthread_mutexattr_init()</i>
6676 <i>pthread_attr_setstack()</i>	<i>pthread_mutexattr_setpshared()</i>
6677 <i>pthread_attr_setstackaddr()</i>	<i>pthread_mutexattr_settype()</i>
6678 <i>pthread_attr_setstacksize()</i>	<i>pthread_once()</i>
6679 <i>pthread_cancel()</i>	<i>pthread_rwlock_destroy()</i>
6680 <i>pthread_cleanup_pop()</i>	<i>pthread_rwlock_init()</i>
6681 <i>pthread_cleanup_push()</i>	<i>pthread_rwlock_rdlock()</i>
6682 <i>pthread_cond_broadcast()</i>	<i>pthread_rwlock_tryrdlock()</i>
6683 <i>pthread_cond_destroy()</i>	<i>pthread_rwlock_trywrlock()</i>
6684 <i>pthread_cond_init()</i>	<i>pthread_rwlock_unlock()</i>
6685 <i>pthread_cond_signal()</i>	<i>pthread_rwlock_wrlock()</i>
6686 <i>pthread_cond_timedwait()</i>	<i>pthread_rwlockattr_destroy()</i>
6687 <i>pthread_cond_wait()</i>	<i>pthread_rwlockattr_getpshared()</i>
6688 <i>pthread_condattr_destroy()</i>	<i>pthread_rwlockattr_init()</i>

6689	<i>pthread_condattr_getpshared()</i>	<i>pthread_rwlockattr_setpshared()</i>
6690	<i>pthread_condattr_init()</i>	<i>pthread_self()</i>
6691	<i>pthread_condattr_setpshared()</i>	<i>pthread_setcancelstate()</i>
6692	<i>pthread_create()</i>	<i>pthread_setcanceltype()</i>
6693	<i>pthread_detach()</i>	<i>pthread_setconcurrency()</i>
6694	<i>pthread_equal()</i>	<i>pthread_setspecific()</i>
6695	<i>pthread_exit()</i>	<i>pthread_sigmask()</i>
6696	<i>pthread_getconcurrency()</i>	<i>pthread_testcancel()</i>
6697	<i>pthread_getspecific()</i>	<i>sigwait()</i>
6698	<i>pthread_join()</i>	

6699 On XSI-conformant systems, the symbolic constant `_POSIX_THREAD_SAFE_FUNCTIONS` is  
 6700 always defined. Therefore, the following functions are always supported:

6701	<i>asctime_r()</i>	<i>getpwuid_r()</i>
6702	<i>ctime_r()</i>	<i>gmtime_r()</i>
6703	<i>flockfile()</i>	<i>localtime_r()</i>
6704	<i>ftrylockfile()</i>	<i>putc_unlocked()</i>
6705	<i>funlockfile()</i>	<i>putchar_unlocked()</i>
6706	<i>getc_unlocked()</i>	<i>rand_r()</i>
6707	<i>getchar_unlocked()</i>	<i>readdir_r()</i>
6708	<i>getgrgid_r()</i>	<i>strerror_r()</i>
6709	<i>getgrnam_r()</i>	<i>strtok_r()</i>
6710	<i>getpwnam_r()</i>	

6711 The following threads functions are only supported on XSI-conformant systems if the Realtime  
 6712 Threads Option Group is supported :

6713	<i>pthread_attr_getinheritsched()</i>	<i>pthread_mutex_getprioceiling()</i>
6714	<i>pthread_attr_getschedpolicy()</i>	<i>pthread_mutex_setprioceiling()</i>
6715	<i>pthread_attr_getscope()</i>	<i>pthread_mutexattr_getprioceiling()</i>
6716	<i>pthread_attr_setinheritsched()</i>	<i>pthread_mutexattr_getprotocol()</i>
6717	<i>pthread_attr_setschedpolicy()</i>	<i>pthread_mutexattr_setprioceiling()</i>
6718	<i>pthread_attr_setscope()</i>	<i>pthread_mutexattr_setprotocol()</i>
6719	<i>pthread_getschedparam()</i>	<i>pthread_setschedparam()</i>

## 6720 XSI Threads Extensions

6721 The following XSI extensions to POSIX.1c are now supported in IEEE Std 1003.1-2001 as part of  
 6722 the alignment with the Single UNIX Specification:

- 6723 • Extended mutex attribute types
- 6724 • Read-write locks and attributes (also introduced by the IEEE Std 1003.1j-2000 amendment)
- 6725 • Thread concurrency level
- 6726 • Thread stack guard size
- 6727 • Parallel I/O

6728 A total of 19 new functions were added.

6729 These extensions carefully follow the threads programming model specified in POSIX.1c. As  
 6730 with POSIX.1c, all the new functions return zero if successful; otherwise, an error number is

returned to indicate the error.

The concept of attribute objects was introduced in POSIX.1c to allow implementations to extend IEEE Std 1003.1-2001 without changing the existing interfaces. Attribute objects were defined for threads, mutexes, and condition variables. Attributes objects are defined as implementation-defined opaque types to aid extensibility, and functions are defined to allow attributes to be set or retrieved. This model has been followed when adding the new type attribute of **pthread\_mutexattr\_t** or the new read-write lock attributes object **pthread\_rwlockattr\_t**.

- Extended Mutex Attributes

POSIX.1c defines a mutex attributes object as an implementation-defined opaque object of type **pthread\_mutexattr\_t**, and specifies a number of attributes which this object must have and a number of functions which manipulate these attributes. These attributes include *detachstate*, *inheritsched*, *schedparm*, *schedpolicy*, *contentionscope*, *stackaddr*, and *stacksize*.

The System Interfaces volume of IEEE Std 1003.1-2001 specifies another mutex attribute called *type*. The *type* attribute allows applications to specify the behavior of mutex locking operations in situations where POSIX.1c behavior is undefined. The OSF DCE threads implementation, based on Draft 4 of POSIX.1c, specified a similar attribute. Note that the names of the attributes have changed somewhat from the OSF DCE threads implementation.

The System Interfaces volume of IEEE Std 1003.1-2001 also extends the specification of the following POSIX.1c functions which manipulate mutexes:

```
pthread_mutex_lock()
pthread_mutex_trylock()
pthread_mutex_unlock()
```

to take account of the new mutex attribute type and to specify behavior which was declared as undefined in POSIX.1c. How a calling thread acquires or releases a mutex now depends upon the mutex *type* attribute.

The *type* attribute can have the following values:

**PTHREAD\_MUTEX\_NORMAL**

Basic mutex with no specific error checking built in. Does not report a deadlock error.

**PTHREAD\_MUTEX\_RECURSIVE**

Allows any thread to recursively lock a mutex. The mutex must be unlocked an equal number of times to release the mutex.

**PTHREAD\_MUTEX\_ERRORCHECK**

Detects and reports simple usage errors; that is, an attempt to unlock a mutex that is not locked by the calling thread or that is not locked at all, or an attempt to relock a mutex the thread already owns.

**PTHREAD\_MUTEX\_DEFAULT**

The default mutex type. May be mapped to any of the above mutex types or may be an implementation-defined type.

*Normal* mutexes do not detect deadlock conditions; for example, a thread will hang if it tries to relock a normal mutex that it already owns. Attempting to unlock a mutex locked by another thread, or unlocking an unlocked mutex, results in undefined behavior. Normal mutexes will usually be the fastest type of mutex available on a platform but provide the least error checking.

*Recursive* mutexes are useful for converting old code where it is difficult to establish clear boundaries of synchronization. A thread can relock a recursive mutex without first unlocking

it. The relocking deadlock which can occur with normal mutexes cannot occur with this type of mutex. However, multiple locks of a recursive mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex. Furthermore, this type of mutex maintains the concept of an owner. Thus, a thread attempting to unlock a recursive mutex which another thread has locked returns with an error. A thread attempting to unlock a recursive mutex that is not locked returns with an error. Never use a recursive mutex with condition variables because the implicit unlock performed by *pthread\_cond\_wait()* or *pthread\_cond\_timedwait()* will not actually release the mutex if it had been locked multiple times.

*Errorcheck* mutexes provide error checking and are useful primarily as a debugging aid. A thread attempting to relock an errorcheck mutex without first unlocking it returns with an error. Again, this type of mutex maintains the concept of an owner. Thus, a thread attempting to unlock an errorcheck mutex which another thread has locked returns with an error. A thread attempting to unlock an errorcheck mutex that is not locked also returns with an error. It should be noted that errorcheck mutexes will almost always be much slower than normal mutexes due to the extra state checks performed.

The default mutex type provides implementation-defined error checking. The default mutex may be mapped to one of the other defined types or may be something entirely different. This enables each vendor to provide the mutex semantics which the vendor feels will be most useful to their target users. Most vendors will probably choose to make normal mutexes the default so as to give applications the benefit of the fastest type of mutexes available on their platform. Check your implementation's documentation.

An application developer can use any of the mutex types almost interchangeably as long as the application does not depend upon the implementation detecting (or failing to detect) any particular errors. Note that a recursive mutex can be used with condition variable waits as long as the application never recursively locks the mutex.

Two functions are provided for manipulating the *type* attribute of a mutex attributes object. This attribute is set or returned in the *type* parameter of these functions. The *pthread\_mutexattr\_settype()* function is used to set a specific type value while *pthread\_mutexattr\_gettype()* is used to return the type of the mutex. Setting the *type* attribute of a mutex attributes object affects only mutexes initialized using that mutex attributes object. Changing the *type* attribute does not affect mutexes previously initialized using that mutex attributes object.

- Read-Write Locks and Attributes

The read-write locks introduced have been harmonized with those in IEEE Std 1003.1j-2000; see also Section B.2.9.6 (on page 177).

Read-write locks (also known as reader-writer locks) allow a thread to exclusively lock some shared data while updating that data, or allow any number of threads to have simultaneous read-only access to the data.

Unlike a mutex, a read-write lock distinguishes between reading data and writing data. A mutex excludes all other threads. A read-write lock allows other threads access to the data, providing no thread is modifying the data. Thus, a read-write lock is less primitive than either a mutex-condition variable pair or a semaphore.

Application developers should consider using a read-write lock rather than a mutex to protect data that is frequently referenced but seldom modified. Most threads (readers) will be able to read the data without waiting and will only have to block when some other thread (a writer) is in the process of modifying the data. Conversely a thread that wants to change the data is forced to wait until there are no readers. This type of lock is often used to facilitate

parallel access to data on multi-processor platforms or to avoid context switches on single processor platforms where multiple threads access the same data.

If a read-write lock becomes unlocked and there are multiple threads waiting to acquire the write lock, the implementation's scheduling policy determines which thread acquires the read-write lock for writing. If there are multiple threads blocked on a read-write lock for both read locks and write locks, it is unspecified whether the readers or a writer acquire the lock first. However, for performance reasons, implementations often favor writers over readers to avoid potential writer starvation.

A read-write lock object is an implementation-defined opaque object of type **pthread\_rwlock\_t** as defined in **<pthread.h>**. There are two different sorts of locks associated with a read-write lock: a read lock and a write lock.

The *pthread\_rwlockattr\_init()* function initializes a read-write lock attributes object with the default value for all the attributes defined in the implementation. After a read-write lock attributes object has been used to initialize one or more read-write locks, changes to the read-write lock attributes object, including destruction, do not affect previously initialized read-write locks.

Implementations must provide at least the read-write lock attribute *process-shared*. This attribute can have the following values:

**PTHREAD\_PROCESS\_SHARED**

Any thread of any process that has access to the memory where the read-write lock resides can manipulate the read-write lock.

**PTHREAD\_PROCESS\_PRIVATE**

Only threads created within the same process as the thread that initialized the read-write lock can manipulate the read-write lock. This is the default value.

The *pthread\_rwlockattr\_setpshared()* function is used to set the *process-shared* attribute of an initialized read-write lock attributes object while the function *pthread\_rwlockattr\_getpshared()* obtains the current value of the *process-shared* attribute.

A read-write lock attributes object is destroyed using the *pthread\_rwlockattr\_destroy()* function. The effect of subsequent use of the read-write lock attributes object is undefined.

A thread creates a read-write lock using the *pthread\_rwlock\_init()* function. The attributes of the read-write lock can be specified by the application developer; otherwise, the default implementation-defined read-write lock attributes are used if the pointer to the read-write lock attributes object is NULL. In cases where the default attributes are appropriate, the **PTHREAD\_RWLOCK\_INITIALIZER** macro can be used to initialize statically allocated read-write locks.

A thread which wants to apply a read lock to the read-write lock can use either *pthread\_rwlock\_rdlock()* or *pthread\_rwlock\_tryrdlock()*. If *pthread\_rwlock\_rdlock()* is used, the thread acquires a read lock if a writer does not hold the write lock and there are no writers blocked on the write lock. If a read lock is not acquired, the calling thread blocks until it can acquire a lock. However, if *pthread\_rwlock\_tryrdlock()* is used, the function returns immediately with the error [EBUSY] if any thread holds a write lock or there are blocked writers waiting for the write lock.

A thread which wants to apply a write lock to the read-write lock can use either of two functions: *pthread\_rwlock\_wrlock()* or *pthread\_rwlock\_trywrlock()*. If *pthread\_rwlock\_wrlock()* is used, the thread acquires the write lock if no other reader or writer threads hold the read-write lock. If the write lock is not acquired, the thread blocks until it can acquire the write lock. However, if *pthread\_rwlock\_trywrlock()* is used, the function returns immediately with

the error [EBUSY] if any thread is holding either a read or a write lock.

The `pthread_rwlock_unlock()` function is used to unlock a read-write lock object held by the calling thread. Results are undefined if the read-write lock is not held by the calling thread. If there are other read locks currently held on the read-write lock object, the read-write lock object remains in the read locked state but without the current thread as one of its owners. If this function releases the last read lock for this read-write lock object, the read-write lock object is put in the unlocked read state. If this function is called to release a write lock for this read-write lock object, the read-write lock object is put in the unlocked state.

- Thread Concurrency Level

On threads implementations that multiplex user threads onto a smaller set of kernel execution entities, the system attempts to create a reasonable number of kernel execution entities for the application upon application startup.

On some implementations, these kernel entities are retained by user threads that block in the kernel. Other implementations do not *timeslice* user threads so that multiple compute-bound user threads can share a kernel thread. On such implementations, some applications may use up all the available kernel execution entities before their user-space threads are used up. The process may be left with user threads capable of doing work for the application but with no way to schedule them.

The `pthread_setconcurrency()` function enables an application to request more kernel entities; that is, specify a desired concurrency level. However, this function merely provides a hint to the implementation. The implementation is free to ignore this request or to provide some other number of kernel entities. If an implementation does not multiplex user threads onto a smaller number of kernel execution entities, the `pthread_setconcurrency()` function has no effect.

The `pthread_setconcurrency()` function may also have an effect on implementations where the kernel mode and user mode schedulers cooperate to ensure that ready user threads are not prevented from running by other threads blocked in the kernel.

The `pthread_getconcurrency()` function always returns the value set by a previous call to `pthread_setconcurrency()`. However, if `pthread_setconcurrency()` was not previously called, this function returns zero to indicate that the threads implementation is maintaining the concurrency level.

- Thread Stack Guard Size

DCE threads introduced the concept of a “thread stack guard size”. Most thread implementations add a region of protected memory to a thread’s stack, commonly known as a “guard region”, as a safety measure to prevent stack pointer overflow in one thread from corrupting the contents of another thread’s stack. The default size of the guard regions attribute is {PAGESIZE} bytes and is implementation-defined.

Some application developers may wish to change the stack guard size. When an application creates a large number of threads, the extra page allocated for each stack may strain system resources. In addition to the extra page of memory, the kernel’s memory manager has to keep track of the different protections on adjoining pages. When this is a problem, the application developer may request a guard size of 0 bytes to conserve system resources by eliminating stack overflow protection.

Conversely an application that allocates large data structures such as arrays on the stack may wish to increase the default guard size in order to detect stack overflow. If a thread allocates two pages for a data array, a single guard page provides little protection against thread stack overflows since the thread can corrupt adjoining memory beyond the guard page.



The System Interfaces volume of IEEE Std 1003.1-2001 defines a new attribute of a thread attributes object; that is, the *guardsize* attribute which allows applications to specify the size of the guard region of a thread's stack.

Two functions are provided for manipulating a thread's stack guard size. The *pthread\_attr\_setguardsize()* function sets the thread *guardsize* attribute, and the *pthread\_attr\_getguardsize()* function retrieves the current value.

An implementation may round up the requested guard size to a multiple of the configurable system variable {PAGESIZE}. In this case, *pthread\_attr\_getguardsize()* returns the guard size specified by the previous *pthread\_attr\_setguardsize()* function call and not the rounded up value.

If an application is managing its own thread stacks using the *stackaddr* attribute, the *guardsize* attribute is ignored and no stack overflow protection is provided. In this case, it is the responsibility of the application to manage stack overflow along with stack allocation.

- Parallel I/O

Suppose two or more threads independently issue read requests on the same file. To read specific data from a file, a thread must first call *lseek()* to seek to the proper offset in the file, and then call *read()* to retrieve the required data. If more than one thread does this at the same time, the first thread may complete its seek call, but before it gets a chance to issue its read call a second thread may complete its seek call, resulting in the first thread accessing incorrect data when it issues its read call. One workaround is to lock the file descriptor while seeking and reading or writing, but this reduces parallelism and adds overhead.

Instead, the System Interfaces volume of IEEE Std 1003.1-2001 provides two functions to make seek/read and seek/write operations atomic. The file descriptor's current offset is unchanged, thus allowing multiple read and write operations to proceed in parallel. This improves the I/O performance of threaded applications. The *pread()* function is used to do an atomic read of data from a file into a buffer. Conversely, the *pwrite()* function does an atomic write of data from a buffer to a file.

### B.2.9.1 Thread-Safety

All functions required by IEEE Std 1003.1-2001 need to be thread-safe. Implementations have to provide internal synchronization when necessary in order to achieve this goal. In certain cases—for example, most floating-point implementations—context switch code may have to manage the writable shared state.

While a read from a pipe of {PIPE\_MAX}\*2 bytes may not generate a single atomic and thread-safe stream of bytes, it should generate “several” (individually atomic) thread-safe streams of bytes. Similarly, while reading from a terminal device may not generate a single atomic and thread-safe stream of bytes, it should generate some finite number of (individually atomic) and thread-safe streams of bytes. That is, concurrent calls to read for a pipe, FIFO, or terminal device are not allowed to result in corrupting the stream of bytes or other internal data. However, *read()*, in these cases, is not required to return a single contiguous and atomic stream of bytes.

It is not required that all functions provided by IEEE Std 1003.1-2001 be either async-cancel-safe or async-signal-safe.

As it turns out, some functions are inherently not thread-safe; that is, their interface specifications preclude reentrancy. For example, some functions (such as *asctime()*) return a pointer to a result stored in memory space allocated by the function on a per-process basis. Such a function is not thread-safe, because its result can be overwritten by successive invocations. Other functions, while not inherently non-thread-safe, may be implemented in ways that lead to

them not being thread-safe. For example, some functions (such as *rand()*) store state information (such as a seed value, which survives multiple function invocations) in memory space allocated by the function on a per-process basis. The implementation of such a function is not thread-safe if the implementation fails to synchronize invocations of the function and thus fails to protect the state information. The problem is that when the state information is not protected, concurrent invocations can interfere with one another (for example, applications using *rand()* may see the same seed value).

#### *Thread-Safety and Locking of Existing Functions*

Originally, POSIX.1 was not designed to work in a multi-threaded environment, and some implementations of some existing functions will not work properly when executed concurrently. To provide routines that will work correctly in an environment with threads (“thread-safe”), two problems need to be solved:

1. Routines that maintain or return pointers to static areas internal to the routine (which may now be shared) need to be modified. The routines *ttymame()* and *localtime()* are examples.
2. Routines that access data space shared by more than one thread need to be modified. The *malloc()* function and the *stdio* family routines are examples.

There are a variety of constraints on these changes. The first is compatibility with the existing versions of these functions—non-thread-safe functions will continue to be in use for some time, as the original interfaces are used by existing code. Another is that the new thread-safe versions of these functions represent as small a change as possible over the familiar interfaces provided by the existing non-thread-safe versions. The new interfaces should be independent of any particular threads implementation. In particular, they should be thread-safe without depending on explicit thread-specific memory. Finally, there should be minimal performance penalty due to the changes made to the functions.

It is intended that the list of functions from POSIX.1 that cannot be made thread-safe and for which corrected versions are provided be complete.

#### *Thread-Safety and Locking Solutions*

Many of the POSIX.1 functions were thread-safe and did not change at all. However, some functions (for example, the math functions typically found in **libm**) are not thread-safe because of writable shared global state. For instance, in IEEE Std 754-1985 floating-point implementations, the computation modes and flags are global and shared.

Some functions are not thread-safe because a particular implementation is not reentrant, typically because of a non-essential use of static storage. These require only a new implementation.

Thread-safe libraries are useful in a wide range of parallel (and asynchronous) programming environments, not just within pthreads. In order to be used outside the context of pthreads, however, such libraries still have to use some synchronization method. These could either be independent of the pthread synchronization operations, or they could be a subset of the pthread interfaces. Either method results in thread-safe library implementations that can be used without the rest of pthreads.

Some functions, such as the *stdio* family interface and dynamic memory allocation functions such as *malloc()*, are inter-dependent routines that share resources (for example, buffers) across related calls. These require synchronization to work correctly, but they do not require any change to their external (user-visible) interfaces.

In some cases, such as *getc()* and *putc()*, adding synchronization is likely to create an unacceptable performance impact. In this case, slower thread-safe synchronized functions are to

be provided, but the original, faster (but unsafe) functions (which may be implemented as macros) are retained under new names. Some additional special-purpose synchronization facilities are necessary for these macros to be usable in multi-threaded programs. This also requires changes in `<stdio.h>`.

The other common reason that functions are unsafe is that they return a pointer to static storage, making the functions non-thread-safe. This has to be changed, and there are three natural choices:

1. Return a pointer to thread-specific storage

This could incur a severe performance penalty on those architectures with a costly implementation of the thread-specific data interface.

A variation on this technique is to use `malloc()` to allocate storage for the function output and return a pointer to this storage. This technique may also have an undesirable performance impact, however, and a simplistic implementation requires that the user program explicitly free the storage object when it is no longer needed. This technique is used by some existing POSIX.1 functions. With careful implementation for infrequently used functions, there may be little or no performance or storage penalty, and the maintenance of already-standardized interfaces is a significant benefit.

2. Return the actual value computed by the function

This technique can only be used with functions that return pointers to structures—routines that return character strings would have to wrap their output in an enclosing structure in order to return the output on the stack. There is also a negative performance impact inherent in this solution in that the output value has to be copied twice before it can be used by the calling function: once from the called routine's local buffers to the top of the stack, then from the top of the stack to the assignment target. Finally, many older compilers cannot support this technique due to a historical tendency to use internal static buffers to deliver the results of structure-valued functions.

3. Have the caller pass the address of a buffer to contain the computed value

The only disadvantage of this approach is that extra arguments have to be provided by the calling program. It represents the most efficient solution to the problem, however, and, unlike the `malloc()` technique, it is semantically clear.

There are some routines (often groups of related routines) whose interfaces are inherently non-thread-safe because they communicate across multiple function invocations by means of static memory locations. The solution is to redesign the calls so that they are thread-safe, typically by passing the needed data as extra parameters. Unfortunately, this may require major changes to the interface as well.

A floating-point implementation using IEEE Std 754-1985 is a case in point. A less problematic example is the `rand48` family of pseudo-random number generators. The functions `getgrgid()`, `getgrnam()`, `getpwnam()`, and `getpwuid()` are another such case.

The problems with `errno` are discussed in **Alternative Solutions for Per-Thread `errno`** (on page 94).

Some functions can be thread-safe or not, depending on their arguments. These include the `tmpnam()` and `ctermid()` functions. These functions have pointers to character strings as arguments. If the pointers are not NULL, the functions store their results in the character string; however, if the pointers are NULL, the functions store their results in an area that may be static and thus subject to overwriting by successive calls. These should only be called by multi-thread applications when their arguments are non-NULL.

### Asynchronous Safety and Thread-Safety

A floating-point implementation has many modes that effect rounding and other aspects of computation. Functions in some math library implementations may change the computation modes for the duration of a function call. If such a function call is interrupted by a signal or cancellation, the floating-point state is not required to be protected.

There is a significant cost to make floating-point operations async-cancel-safe or async-signal-safe; accordingly, neither form of async safety is required.

### Functions Returning Pointers to Static Storage

For those functions that are not thread-safe because they return values in fixed size statically allocated structures, alternate “\_r” forms are provided that pass a pointer to an explicit result structure. Those that return pointers into library-allocated buffers have forms provided with explicit buffer and length parameters.

For functions that return pointers to library-allocated buffers, it makes sense to provide “\_r” versions that allow the application control over allocation of the storage in which results are returned. This allows the state used by these functions to be managed on an application-specific basis, supporting per-thread, per-process, or other application-specific sharing relationships.

Early proposals had provided “\_r” versions for functions that returned pointers to variable-size buffers without providing a means for determining the required buffer size. This would have made using such functions exceedingly clumsy, potentially requiring iteratively calling them with increasingly larger guesses for the amount of storage required. Hence, *sysconf()* variables have been provided for such functions that return the maximum required buffer size.

Thus, the rule that has been followed by IEEE Std 1003.1-2001 when adapting single-threaded non-thread-safe functions is as follows: all functions returning pointers to library-allocated storage should have “\_r” versions provided, allowing the application control over the storage allocation. Those with variable-sized return values accept both a buffer address and a length parameter. The *sysconf()* variables are provided to supply the appropriate buffer sizes when required. Implementors are encouraged to apply the same rule when adapting their own existing functions to a pthreads environment.

### B.2.9.2 Thread IDs

Separate applications should communicate through well-defined interfaces and should not depend on each other's implementation. For example, if a programmer decides to rewrite the *sort* utility using multiple threads, it should be easy to do this so that the interface to the *sort* utility does not change. Consider that if the user causes SIGINT to be generated while the *sort* utility is running, keeping the same interface means that the entire *sort* utility is killed, not just one of its threads. As another example, consider a realtime application that manages a reactor. Such an application may wish to allow other applications to control the priority at which it watches the control rods. One technique to accomplish this is to write the ID of the thread watching the control rods into a file and allow other programs to change the priority of that thread as they see fit. A simpler technique is to have the reactor process accept IPCs (Interprocess Communication messages) from other processes, telling it at a semantic level what priority the program should assign to watching the control rods. This allows the programmer greater flexibility in the implementation. For example, the programmer can change the implementation from having one thread per rod to having one thread watching all of the rods without changing the interface. Having threads live inside the process means that the implementation of a process is invisible to outside processes (excepting debuggers and system management tools).

Threads do not provide a protection boundary. Every thread model allows threads to share memory with other threads and encourages this sharing to be widespread. This means that one

7103 thread can wipe out memory that is needed for the correct functioning of other threads that are  
 7104 sharing its memory. Consequently, providing each thread with its own user and/or group IDs  
 7105 would not provide a protection boundary between threads sharing memory.

### 7106 B.2.9.3 Thread Mutexes

7107 There is no additional rationale provided for this section.

### 7108 B.2.9.4 Thread Scheduling

#### 7109 • Scheduling Implementation Models

7110 The following scheduling implementation models are presented in terms of threads and  
 7111 “kernel entities”. This is to simplify exposition of the models, and it does not imply that an  
 7112 implementation actually has an identifiable “kernel entity”.

7113 A kernel entity is not defined beyond the fact that it has scheduling attributes that are used to  
 7114 resolve contention with other kernel entities for execution resources. A kernel entity may be  
 7115 thought of as an envelope that holds a thread or a separate kernel thread. It is not a  
 7116 conventional process, although it shares with the process the attribute that it has a single  
 7117 thread of control; it does not necessarily imply an address space, open files, and so on. It is  
 7118 better thought of as a primitive facility upon which conventional processes and threads may  
 7119 be constructed.

#### 7120 — System Thread Scheduling Model

7121 This model consists of one thread per kernel entity. The kernel entity is solely responsible  
 7122 for scheduling thread execution on one or more processors. This model schedules all  
 7123 threads against all other threads in the system using the scheduling attributes of the  
 7124 thread.

#### 7125 — Process Scheduling Model

7126 A generalized process scheduling model consists of two levels of scheduling. A threads  
 7127 library creates a pool of kernel entities, as required, and schedules threads to run on them  
 7128 using the scheduling attributes of the threads. Typically, the size of the pool is a function  
 7129 of the simultaneously runnable threads, not the total number of threads. The kernel then  
 7130 schedules the kernel entities onto processors according to their scheduling attributes,  
 7131 which are managed by the threads library. This set model potentially allows a wide range  
 7132 of mappings between threads and kernel entities.

#### 7133 • System and Process Scheduling Model Performance

7134 There are a number of important implications on the performance of applications using these  
 7135 scheduling models. The process scheduling model potentially provides lower overhead for  
 7136 making scheduling decisions, since there is no need to access kernel-level information or  
 7137 functions and the set of schedulable entities is smaller (only the threads within the process).

7138 On the other hand, since the kernel is also making scheduling decisions regarding the system  
 7139 resources under its control (for example, CPU(s), I/O devices, memory), decisions that do  
 7140 not take thread scheduling parameters into account can result in unspecified delays for  
 7141 realtime application threads, causing them to miss maximum response time limits.

#### 7142 • Rate Monotonic Scheduling

7143 Rate monotonic scheduling was considered, but rejected for standardization in the context of  
 7144 pthreads. A sporadic server policy is included.

- Scheduling Options

In IEEE Std 1003.1-2001, the basic thread scheduling functions are defined under the Threads option, so that they are required of all threads implementations. However, there are no specific scheduling policies required by this option to allow for conforming thread implementations that are not targeted to realtime applications.

Specific standard scheduling policies are defined to be under the Thread Execution Scheduling option, and they are specifically designed to support realtime applications by providing predictable resource-sharing sequences. The name of this option was chosen to emphasize that this functionality is defined as appropriate for realtime applications that require simple priority-based scheduling.

It is recognized that these policies are not necessarily satisfactory for some multi-processor implementations, and work is ongoing to address a wider range of scheduling behaviors. The interfaces have been chosen to create abundant opportunity for future scheduling policies to be implemented and standardized based on this interface. In order to standardize a new scheduling policy, all that is required (from the standpoint of thread scheduling attributes) is to define a new policy name, new members of the thread attributes object, and functions to set these members when the scheduling policy is equal to the new value.

### Scheduling Contention Scope

In order to accommodate the requirement for realtime response, each thread has a scheduling contention scope attribute. Threads with a system scheduling contention scope have to be scheduled with respect to all other threads in the system. These threads are usually bound to a single kernel entity that reflects their scheduling attributes and are directly scheduled by the kernel.

Threads with a process scheduling contention scope need be scheduled only with respect to the other threads in the process. These threads may be scheduled within the process onto a pool of kernel entities. The implementation is also free to bind these threads directly to kernel entities and let them be scheduled by the kernel. Process scheduling contention scope allows the implementation the most flexibility and is the default if both contention scopes are supported and none is specified.

Thus, the choice by implementors to provide one or the other (or both) of these scheduling models is driven by the need of their supported application domains for worst-case (that is, realtime) response, or average-case (non-realtime) response.

### Scheduling Allocation Domain

The SCHED\_FIFO and SCHED\_RR scheduling policies take on different characteristics on a multi-processor. Other scheduling policies are also subject to changed behavior when executed on a multi-processor. The concept of scheduling allocation domain determines the set of processors on which the threads of an application may run. By considering the application's processor scheduling allocation domain for its threads, scheduling policies can be defined in terms of their behavior for varying processor scheduling allocation domain values. It is conceivable that not all scheduling allocation domain sizes make sense for all scheduling policies on all implementations. The concept of scheduling allocation domain, however, is a useful tool for the description of multi-processor scheduling policies.

The “process control” approach to scheduling obtains significant performance advantages from dynamic scheduling allocation domain sizes when it is applicable.

Non-Uniform Memory Access (NUMA) multi-processors may use a system scheduling structure that involves reassignment of threads among scheduling allocation domains. In NUMA

7191 machines, a natural model of scheduling is to match scheduling allocation domains to clusters of  
7192 processors. Load balancing in such an environment requires changing the scheduling allocation  
7193 domain to which a thread is assigned.

#### 7194 **Scheduling Documentation**

7195 Implementation-provided scheduling policies need to be completely documented in order to be  
7196 useful. This documentation includes a description of the attributes required for the policy, the  
7197 scheduling interaction of threads running under this policy and all other supported policies, and  
7198 the effects of all possible values for processor scheduling allocation domain. Note that for the  
7199 implementor wishing to be minimally-compliant, it is (minimally) acceptable to define the  
7200 behavior as undefined.

#### 7201 **Scheduling Contention Scope Attribute**

7202 The scheduling contention scope defines how threads compete for resources. Within  
7203 IEEE Std 1003.1-2001, scheduling contention scope is used to describe only how threads are  
7204 scheduled in relation to one another in the system. That is, either they are scheduled against all  
7205 other threads in the system ("system scope") or only against those threads in the process  
7206 ("process scope"). In fact, scheduling contention scope may apply to additional resources,  
7207 including virtual timers and profiling, which are not currently considered by  
7208 IEEE Std 1003.1-2001.

#### 7209 **Mixed Scopes**

7210 If only one scheduling contention scope is supported, the scheduling decision is straightforward.  
7211 To perform the processor scheduling decision in a mixed scope environment, it is necessary to  
7212 map the scheduling attributes of the thread with process-wide contention scope to the same  
7213 attribute space as the thread with system-wide contention scope.

7214 Since a conforming implementation has to support one and may support both scopes, it is useful  
7215 to discuss the effects of such choices with respect to example applications. If an implementation  
7216 supports both scopes, mixing scopes provides a means of better managing system-level (that is,  
7217 kernel-level) and library-level resources. In general, threads with system scope will require the  
7218 resources of a separate kernel entity in order to guarantee the scheduling semantics. On the  
7219 other hand, threads with process scope can share the resources of a kernel entity while  
7220 maintaining the scheduling semantics.

7221 The application is free to create threads with dedicated kernel resources, and other threads that  
7222 multiplex kernel resources. Consider the example of a window server. The server allocates two  
7223 threads per widget: one thread manages the widget user interface (including drawing), while the  
7224 other thread takes any required application action. This allows the widget to be "active" while  
7225 the application is computing. A screen image may be built from thousands of widgets. If each of  
7226 these threads had been created with system scope, then most of the kernel-level resources might  
7227 be wasted, since only a few widgets are active at any one time. In addition, mixed scope is  
7228 particularly useful in a window server where one thread with high priority and system scope  
7229 handles the mouse so that it tracks well. As another example, consider a database server. For  
7230 each of the hundreds or thousands of clients supported by a large server, an equivalent number  
7231 of threads will have to be created. If each of these threads were system scope, the consequences  
7232 would be the same as for the window server example above. However, the server could be  
7233 constructed so that actual retrieval of data is done by several dedicated threads. Dedicated  
7234 threads that do work for all clients frequently justify the added expense of system scope. If it  
7235 were not permissible to mix system and process threads in the same process, this type of  
7236 solution would not be possible.

**Dynamic Thread Scheduling Parameters Access**

In many time-constrained applications, there is no need to change the scheduling attributes dynamically during thread or process execution, since the general use of these attributes is to reflect directly the time constraints of the application. Since these time constraints are generally imposed to meet higher-level system requirements, such as accuracy or availability, they frequently should remain unchanged during application execution.

However, there are important situations in which the scheduling attributes should be changed. Generally, this will occur when external environmental conditions exist in which the time constraints change. Consider, for example, a space vehicle major mode change, such as the change from ascent to descent mode, or the change from the space environment to the atmospheric environment. In such cases, the frequency with which many of the sensors or actuators need to be read or written will change, which will necessitate a priority change. In other cases, even the existence of a time constraint might be temporary, necessitating not just a priority change, but also a policy change for ongoing threads or processes. For this reason, it is critical that the interface should provide functions to change the scheduling parameters dynamically, but, as with many of the other realtime functions, it is important that applications use them properly to avoid the possibility of unnecessarily degrading performance.

In providing functions for dynamically changing the scheduling behavior of threads, there were two options: provide functions to get and set the individual scheduling parameters of threads, or provide a single interface to get and set all the scheduling parameters for a given thread simultaneously. Both approaches have merit. Access functions for individual parameters allow simpler control of thread scheduling for simple thread scheduling parameters. However, a single function for setting all the parameters for a given scheduling policy is required when first setting that scheduling policy. Since the single all-encompassing functions are required, it was decided to leave the interface as minimal as possible. Note that simpler functions (such as *pthread\_setprio()* for threads running under the priority-based schedulers) can be easily defined in terms of the all-encompassing functions.

If the *pthread\_setschedparam()* function executes successfully, it will have set all of the scheduling parameter values indicated in *param*; otherwise, none of the scheduling parameters will have been modified. This is necessary to ensure that the scheduling of this and all other threads continues to be consistent in the presence of an erroneous scheduling parameter.

The [EPERM] error value is included in the list of possible *pthread\_setschedparam()* error returns as a reflection of the fact that the ability to change scheduling parameters increases risks to the implementation and application performance if the scheduling parameters are changed improperly. For this reason, and based on some existing practice, it was felt that some implementations would probably choose to define specific permissions for changing either a thread's own or another thread's scheduling parameters. IEEE Std 1003.1-2001 does not include portable methods for setting or retrieving permissions, so any such use of permissions is completely unspecified.

**Mutex Initialization Scheduling Attributes**

In a priority-driven environment, a direct use of traditional primitives like mutexes and condition variables can lead to unbounded priority inversion, where a higher priority thread can be blocked by a lower priority thread, or set of threads, for an unbounded duration of time. As a result, it becomes impossible to guarantee thread deadlines. Priority inversion can be bounded and minimized by the use of priority inheritance protocols. This allows thread deadlines to be guaranteed even in the presence of synchronization requirements.

Two useful but simple members of the family of priority inheritance protocols are the basic priority inheritance protocol and the priority ceiling protocol emulation. Under the Basic Priority



Inheritance protocol (governed by the Thread Priority Inheritance option), a thread that is blocking higher priority threads executes at the priority of the highest priority thread that it blocks. This simple mechanism allows priority inversion to be bounded by the duration of critical sections and makes timing analysis possible.

Under the Priority Ceiling Protocol Emulation protocol (governed by the Thread Priority Protection option), each mutex has a priority ceiling, usually defined as the priority of the highest priority thread that can lock the mutex. When a thread is executing inside critical sections, its priority is unconditionally increased to the highest of the priority ceilings of all the mutexes owned by the thread. This protocol has two very desirable properties in uni-processor systems. First, a thread can be blocked by a lower priority thread for at most the duration of one single critical section. Furthermore, when the protocol is correctly used in a single processor, and if threads do not become blocked while owning mutexes, mutual deadlocks are prevented.

The priority ceiling emulation can be extended to multiple processor environments, in which case the values of the priority ceilings will be assigned depending on the kind of mutex that is being used: local to only one processor, or global, shared by several processors. Local priority ceilings will be assigned the usual way, equal to the priority of the highest priority thread that may lock that mutex. Global priority ceilings will usually be assigned a priority level higher than all the priorities assigned to any of the threads that reside in the involved processors to avoid the effect called remote blocking.

#### **Change the Priority Ceiling of a Mutex**

In order for the priority protect protocol to exhibit its desired properties of bounding priority inversion and avoidance of deadlock, it is critical that the ceiling priority of a mutex be the same as the priority of the highest thread that can ever hold it, or higher. Thus, if the priorities of the threads using such mutexes never change dynamically, there is no need ever to change the priority ceiling of a mutex.

However, if a major system mode change results in an altered response time requirement for one or more application threads, their priority has to change to reflect it. It will occasionally be the case that the priority ceilings of mutexes held also need to change. While changing priority ceilings should generally be avoided, it is important that IEEE Std 1003.1-2001 provide these interfaces for those cases in which it is necessary.

#### **B.2.9.5 Thread Cancellation**

Many existing threads packages have facilities for canceling an operation or canceling a thread. These facilities are used for implementing user requests (such as the CANCEL button in a window-based application), for implementing OR parallelism (for example, telling the other threads to stop working once one thread has found a forced mate in a parallel chess program), or for implementing the ABORT mechanism in Ada.

POSIX programs traditionally have used the signal mechanism combined with either *longjmp()* or polling to cancel operations. Many POSIX programmers have trouble using these facilities to solve their problems efficiently in a single-threaded process. With the introduction of threads, these solutions become even more difficult to use.

The main issues with implementing a cancellation facility are specifying the operation to be canceled, cleanly releasing any resources allocated to that operation, controlling when the target notices that it has been canceled, and defining the interaction between asynchronous signals and cancellation.

**Specifying the Operation to Cancel**

Consider a thread that calls through five distinct levels of program abstraction and then, inside the lowest-level abstraction, calls a function that suspends the thread. (An abstraction boundary is a layer at which the client of the abstraction sees only the service being provided and can remain ignorant of the implementation. Abstractions are often layered, each level of abstraction being a client of the lower-level abstraction and implementing a higher-level abstraction.) Depending on the semantics of each abstraction, one could imagine wanting to cancel only the call that causes suspension, only the bottom two levels, or the operation being done by the entire thread. Canceling operations at a finer grain than the entire thread is difficult because threads are active and they may be run in parallel on a multi-processor. By the time one thread can make a request to cancel an operation, the thread performing the operation may have completed that operation and gone on to start another operation whose cancellation is not desired. Thread IDs are not reused until the thread has exited, and either it was created with the *Attr detachstate* attribute set to *PTHREAD\_CREATE\_DETACHED* or the *pthread\_join()* or *pthread\_detach()* function has been called for that thread. Consequently, a thread cancellation will never be misdirected when the thread terminates. For these reasons, the canceling of operations is done at the granularity of the thread. Threads are designed to be inexpensive enough so that a separate thread may be created to perform each separately cancelable operation; for example, each possibly long running user request.

For cancellation to be used in existing code, cancellation scopes and handlers will have to be established for code that needs to release resources upon cancellation, so that it follows the programming discipline described in the text.

**A Special Signal Versus a Special Interface**

Two different mechanisms were considered for providing the cancellation interfaces. The first was to provide an interface to direct signals at a thread and then to define a special signal that had the required semantics. The other alternative was to use a special interface that delivered the correct semantics to the target thread.

The solution using signals produced a number of problems. It required the implementation to provide cancellation in terms of signals whereas a perfectly valid (and possibly more efficient) implementation could have both layered on a low-level set of primitives. There were so many exceptions to the special signal (it cannot be used with *kill()*, no POSIX.1 interfaces can be used with it) that it was clearly not a valid signal. Its semantics on delivery were also completely different from any existing POSIX.1 signal. As such, a special interface that did not mandate the implementation and did not confuse the semantics of signals and cancellation was felt to be the better solution.

**Races Between Cancellation and Resuming Execution**

Due to the nature of cancellation, there is generally no synchronization between the thread requesting the cancellation of a blocked thread and events that may cause that thread to resume execution. For this reason, and because excess serialization hurts performance, when both an event that a thread is waiting for has occurred and a cancellation request has been made and cancellation is enabled, IEEE Std 1003.1-2001 explicitly allows the implementation to choose between returning from the blocking call or acting on the cancellation request.

## Interaction of Cancellation with Asynchronous Signals

A typical use of cancellation is to acquire a lock on some resource and to establish a cancellation cleanup handler for releasing the resource when and if the thread is canceled.

A correct and complete implementation of cancellation in the presence of asynchronous signals requires considerable care. An implementation has to push a cancellation cleanup handler on the cancellation cleanup stack while maintaining the integrity of the stack data structure. If an asynchronously-generated signal is posted to the thread during a stack operation, the signal handler cannot manipulate the cancellation cleanup stack. As a consequence, asynchronous signal handlers may not cancel threads or otherwise manipulate the cancellation state of a thread. Threads may, of course, be canceled by another thread that used a *sigwait()* function to wait synchronously for an asynchronous signal.

In order for cancellation to function correctly, it is required that asynchronous signal handlers not change the cancellation state. This requires that some elements of existing practice, such as using *longjmp()* to exit from an asynchronous signal handler implicitly, be prohibited in cases where the integrity of the cancellation state of the interrupt thread cannot be ensured.

## Thread Cancellation Overview

### • Cancelability States

The three possible cancelability states (disabled, deferred, and asynchronous) are encoded into two separate bits ((disable, enable) and (deferred, asynchronous)) to allow them to be changed and restored independently. For instance, short code sequences that will not block sometimes disable cancelability on entry and restore the previous state upon exit. Likewise, long or unbounded code sequences containing no convenient explicit cancellation points will sometimes set the cancelability type to asynchronous on entry and restore the previous value upon exit.

### • Cancellation Points

Cancellation points are points inside of certain functions where a thread has to act on any pending cancellation request when cancelability is enabled, if the function would block. As with checking for signals, operations need only check for pending cancellation requests when the operation is about to block indefinitely.

The idea was considered of allowing implementations to define whether blocking calls such as *read()* should be cancellation points. It was decided that it would adversely affect the design of conforming applications if blocking calls were not cancellation points because threads could be left blocked in an uncancelable state.

There are several important blocking routines that are specifically not made cancellation points:

— *pthread\_mutex\_lock()*

If *pthread\_mutex\_lock()* were a cancellation point, every routine that called it would also become a cancellation point (that is, any routine that touched shared state would automatically become a cancellation point). For example, *malloc()*, *free()*, and *rand()* would become cancellation points under this scheme. Having too many cancellation points makes programming very difficult, leading to either much disabling and restoring of cancelability or much difficulty in trying to arrange for reliable cleanup at every possible place.

Since *pthread\_mutex\_lock()* is not a cancellation point, threads could result in being blocked uninterruptibly for long periods of time if mutexes were used as a general

7416 synchronization mechanism. As this is normally not acceptable, mutexes should only be  
 7417 used to protect resources that are held for small fixed lengths of time where not being  
 7418 able to be canceled will not be a problem. Resources that need to be held exclusively for  
 7419 long periods of time should be protected with condition variables.

7420 — *pthread\_barrier\_wait()*

7421 Canceling a barrier wait will render a barrier unusable. Similar to a barrier timeout (which  
 7422 the standard developers rejected), there is no way to guarantee the consistency of a  
 7423 barrier's internal data structures if a barrier wait is canceled.

7424 — *pthread\_spin\_lock()*

7425 As with mutexes, spin locks should only be used to protect resources that are held for  
 7426 small fixed lengths of time where not being cancelable will not be a problem.

7427 Every library routine should specify whether or not it includes any cancellation points.  
 7428 Typically, only those routines that may block or compute indefinitely need to include  
 7429 cancellation points.

7430 Correctly coded routines only reach cancellation points after having set up a cancellation  
 7431 cleanup handler to restore invariants if the thread is canceled at that point. Being cancelable  
 7432 only at specified cancellation points allows programmers to keep track of actions needed in a  
 7433 cancellation cleanup handler more easily. A thread should only be made asynchronously  
 7434 cancelable when it is not in the process of acquiring or releasing resources or otherwise in a  
 7435 state from which it would be difficult or impossible to recover.

#### 7436 • Thread Cancellation Cleanup Handlers

7437 The cancellation cleanup handlers provide a portable mechanism, easy to implement, for  
 7438 releasing resources and restoring invariants. They are easier to use than signal handlers  
 7439 because they provide a stack of cancellation cleanup handlers rather than a single handler,  
 7440 and because they have an argument that can be used to pass context information to the  
 7441 handler.

7442 The alternative to providing these simple cancellation cleanup handlers (whose only use is  
 7443 for cleaning up when a thread is canceled) is to define a general exception package that could  
 7444 be used for handling and cleaning up after hardware traps and software-detected errors. This  
 7445 was too far removed from the charter of providing threads to handle asynchrony. However,  
 7446 it is an explicit goal of IEEE Std 1003.1-2001 to be compatible with existing exception facilities  
 7447 and languages having exceptions.

7448 The interaction of this facility and other procedure-based or language-level exception  
 7449 facilities is unspecified in this version of IEEE Std 1003.1-2001. However, it is intended that it  
 7450 be possible for an implementation to define the relationship between these cancellation  
 7451 cleanup handlers and Ada, C++, or other language-level exception handling facilities.

7452 It was suggested that the cancellation cleanup handlers should also be called when the  
 7453 process exits or calls the *exec* function. This was rejected partly due to the performance  
 7454 problem caused by having to call the cancellation cleanup handlers of every thread before the  
 7455 operation could continue. The other reason was that the only state expected to be cleaned up  
 7456 by the cancellation cleanup handlers would be the intraprocess state. Any handlers that are  
 7457 to clean up the interprocess state would be registered with *atexit()*. There is the orthogonal  
 7458 problem that the *exec* functions do not honor the *atexit()* handlers, but resolving this is  
 7459 beyond the scope of IEEE Std 1003.1-2001.

7460	• Async-Cancel Safety	
7461	A function is said to be async-cancel-safe if it is written in such a way that entering the	
7462	function with asynchronous cancelability enabled will not cause any invariants to be	
7463	violated, even if a cancellation request is delivered at any arbitrary instruction. Functions that	
7464	are async-cancel-safe are often written in such a way that they need to acquire no resources	
7465	for their operation and the visible variables that they may write are strictly limited.	
7466	Any routine that gets a resource as a side effect cannot be made async-cancel-safe (for	
7467	example, <i>malloc()</i> ). If such a routine were called with asynchronous cancelability enabled, it	
7468	might acquire the resource successfully, but as it was returning to the client, it could act on a	
7469	cancellation request. In such a case, the application would have no way of knowing whether	
7470	the resource was acquired or not.	
7471	Indeed, because many interesting routines cannot be made async-cancel-safe, most library	
7472	routines in general are not async-cancel-safe. Every library routine should specify whether or	
7473	not it is async-cancel safe so that programmers know which routines can be called from code	
7474	that is asynchronously cancelable.	
7475	IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/8 is applied, adding the <i>pselect()</i> function	1
7476	to the list of functions with cancellation points.	1
7477	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/5 is applied, adding the <i>fdatsync()</i>	2
7478	function into the table of functions that shall have cancellation points.	2
7479	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/6 is applied, adding the numerous	2
7480	functions into the table of functions that may have cancellation points.	2
7481	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/7 is applied, clarifying the requirements	2
7482	in Thread Cancellation Cleanup Handlers.	2
7483	<b>B.2.9.6 Thread Read-Write Locks</b>	
7484	<b>Background</b>	
7485	Read-write locks are often used to allow parallel access to data on multi-processors, to avoid	
7486	context switches on uni-processors when multiple threads access the same data, and to protect	
7487	data structures that are frequently accessed (that is, read) but rarely updated (that is, written).	
7488	The in-core representation of a file system directory is a good example of such a data structure.	
7489	One would like to achieve as much concurrency as possible when searching directories, but limit	
7490	concurrent access when adding or deleting files.	
7491	Although read-write locks can be implemented with mutexes and condition variables, such	
7492	implementations are significantly less efficient than is possible. Therefore, this synchronization	
7493	primitive is included in IEEE Std 1003.1-2001 for the purpose of allowing more efficient	
7494	implementations in multi-processor systems.	
7495	<b>Queuing of Waiting Threads</b>	
7496	The <i>pthread_rwlock_unlock()</i> function description states that one writer or one or more readers	
7497	must acquire the lock if it is no longer held by any thread as a result of the call. However, the	
7498	function does not specify which thread(s) acquire the lock, unless the Thread Execution	
7499	Scheduling option is supported.	
7500	The standard developers considered the issue of scheduling with respect to the queuing of	
7501	threads blocked on a read-write lock. The question turned out to be whether	
7502	IEEE Std 1003.1-2001 should require priority scheduling of read-write locks for threads whose	

execution scheduling policy is priority-based (for example, SCHED\_FIFO or SCHED\_RR). There are tradeoffs between priority scheduling, the amount of concurrency achievable among readers, and the prevention of writer and/or reader starvation.

For example, suppose one or more readers hold a read-write lock and the following threads request the lock in the listed order:

```
pthread_rwlock_wrlock() - Low priority thread writer_a
pthread_rwlock_rdlock() - High priority thread reader_a
pthread_rwlock_rdlock() - High priority thread reader_b
pthread_rwlock_rdlock() - High priority thread reader_c
```

When the lock becomes available, should *writer\_a* block the high priority readers? Or, suppose a read-write lock becomes available and the following are queued:

```
pthread_rwlock_rdlock() - Low priority thread reader_a
pthread_rwlock_rdlock() - Low priority thread reader_b
pthread_rwlock_rdlock() - Low priority thread reader_c
pthread_rwlock_wrlock() - Medium priority thread writer_a
pthread_rwlock_rdlock() - High priority thread reader_d
```

If priority scheduling is applied then *reader\_d* would acquire the lock and *writer\_a* would block the remaining readers. But should the remaining readers also acquire the lock to increase concurrency? The solution adopted takes into account that when the Thread Execution Scheduling option is supported, high priority threads may in fact starve low priority threads (the application developer is responsible in this case for designing the system in such a way that this starvation is avoided). Therefore, IEEE Std 1003.1-2001 specifies that high priority readers take precedence over lower priority writers. However, to prevent writer starvation from threads of the same or lower priority, writers take precedence over readers of the same or lower priority.

Priority inheritance mechanisms are non-trivial in the context of read-write locks. When a high priority writer is forced to wait for multiple readers, for example, it is not clear which subset of the readers should inherit the writer's priority. Furthermore, the internal data structures that record the inheritance must be accessible to all readers, and this implies some sort of serialization that could negate any gain in parallelism achieved through the use of multiple readers in the first place. Finally, existing practice does not support the use of priority inheritance for read-write locks. Therefore, no specification of priority inheritance or priority ceiling is attempted. If reliable priority-scheduled synchronization is absolutely required, it can always be obtained through the use of mutexes.

### Comparison to *fcntl()* Locks

The read-write locks and the *fcntl()* locks in IEEE Std 1003.1-2001 share a common goal: increasing concurrency among readers, thus increasing throughput and decreasing delay.

However, the read-write locks have two features not present in the *fcntl()* locks. First, under priority scheduling, read-write locks are granted in priority order. Second, also under priority scheduling, writer starvation is prevented by giving writers preference over readers of equal or lower priority.

Also, read-write locks can be used in systems lacking a file system, such as those conforming to the minimal realtime system profile of IEEE Std 1003.13-1998.

7545 **History of Resolution Issues**

7546 Based upon some balloting objections, early drafts specified the behavior of threads waiting on a  
 7547 read-write lock during the execution of a signal handler, as if the thread had not called the lock  
 7548 operation. However, this specified behavior would require implementations to establish  
 7549 internal signal handlers even though this situation would be rare, or never happen for many  
 7550 programs. This would introduce an unacceptable performance hit in comparison to the little  
 7551 additional functionality gained. Therefore, the behavior of read-write locks and signals was  
 7552 reverted back to its previous mutex-like specification.

7553 **B.2.9.7 Thread Interactions with Regular File Operations**

7554 There is no additional rationale provided for this section.

7555 **B.2.9.8 Use of Application-Managed Thread Stacks**

2

7556 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/8 is applied, adding this new section. It 2  
 7557 was added to make it clear that the current standard does not allow an application to determine 2  
 7558 when a stack can be reclaimed. This may be addressed in a future revision. 2

7559 **B.2.10 Sockets**

7560 The base document for the sockets interfaces in IEEE Std 1003.1-2001 is the XNS, Issue 5.2  
 7561 specification. This was primarily chosen as it aligns with IPv6. Additional material has been  
 7562 added from IEEE Std 1003.1g-2000, notably socket concepts, raw sockets, the *pselect()* function,  
 7563 the *socketmark()* function, and the `<sys/select.h>` header.

7564 **B.2.10.1 Address Families**

7565 There is no additional rationale provided for this section.

7566 **B.2.10.2 Addressing**

7567 There is no additional rationale provided for this section.

7568 **B.2.10.3 Protocols**

7569 There is no additional rationale provided for this section.

7570 **B.2.10.4 Routing**

7571 There is no additional rationale provided for this section.

7572 **B.2.10.5 Interfaces**

7573 There is no additional rationale provided for this section.

7574 **B.2.10.6 Socket Types**

7575 The type **socklen\_t** was invented to cover the range of implementations seen in the field. The  
 7576 intent of **socklen\_t** is to be the type for all lengths that are naturally bounded in size; that is, that  
 7577 they are the length of a buffer which cannot sensibly become of massive size; network addresses,  
 7578 host names, string representations of these, ancillary data, control messages, and socket options  
 7579 are examples. Truly boundless sizes are represented by **size\_t** as in *read()*, *write()*, and so on.

7580 All **socklen\_t** types were originally (in BSD UNIX) of type **int**. During the development of  
 7581 IEEE Std 1003.1-2001, it was decided to change all buffer lengths to **size\_t**, which appears at face  
 7582 value to make sense. When dual mode 32/64-bit systems came along, this choice unnecessarily

7583 complicated system interfaces because **size\_t** (with **long**) was a different size under ILP32 and  
7584 LP64 models. Reverting to **int** would have happened except that some implementations had  
7585 already shipped 64-bit-only interfaces. The compromise was a type which could be defined to be  
7586 any size by the implementation: **socklen\_t**.

7587 *B.2.10.7 Socket I/O Mode*

7588 There is no additional rationale provided for this section.

7589 *B.2.10.8 Socket Owner*

7590 There is no additional rationale provided for this section.

7591 *B.2.10.9 Socket Queue Limits*

7592 There is no additional rationale provided for this section.

7593 *B.2.10.10 Pending Error*

7594 There is no additional rationale provided for this section.

7595 *B.2.10.11 Socket Receive Queue*

7596 There is no additional rationale provided for this section.

7597 *B.2.10.12 Socket Out-of-Band Data State*

7598 There is no additional rationale provided for this section.

7599 *B.2.10.13 Connection Indication Queue*

7600 There is no additional rationale provided for this section.

7601 *B.2.10.14 Signals*

7602 There is no additional rationale provided for this section.

7603 *B.2.10.15 Asynchronous Errors*

7604 There is no additional rationale provided for this section.

7605 *B.2.10.16 Use of Options*

7606 There is no additional rationale provided for this section.

7607 *B.2.10.17 Use of Sockets for Local UNIX Connections*

7608 There is no additional rationale provided for this section.

7609 *B.2.10.18 Use of Sockets over Internet Protocols*

7610 A raw socket allows privileged users direct access to a protocol; for example, raw access to the  
7611 IP and ICMP protocols is possible through raw sockets. Raw sockets are intended for  
7612 knowledgeable applications that wish to take advantage of some protocol feature not directly  
7613 accessible through the other sockets interfaces.



7614 *B.2.10.19 Use of Sockets over Internet Protocols Based on IPv4*

7615 There is no additional rationale provided for this section.

7616 *B.2.10.20 Use of Sockets over Internet Protocols Based on IPv6*7617 The Open Group Base Resolution bwg2001-012 is applied, clarifying that IPv6 implementations  
7618 are required to support use of AF\_INET6 sockets over IPv4.7619 **B.2.11 Tracing**7620 The organization of the tracing rationale differs from the traditional rationale in that this tracing  
7621 rationale text is written against the trace interface as a whole, rather than against the individual  
7622 components of the trace interface or the normative section in which those components are  
7623 defined. Therefore the sections below do not parallel the sections of normative text in  
7624 IEEE Std 1003.1-2001.7625 *B.2.11.1 Objectives*7626 The intended uses of tracing are application-system debugging during system development, as a  
7627 “flight recorder” for maintenance of fielded systems, and as a performance measurement tool. In  
7628 all of these intended uses, the vendor-supplied computer system and its software are, for this  
7629 discussion, assumed error-free; the intent being to debug the user-written and/or third-party  
7630 application code, and their interactions. Clearly, problems with the vendor-supplied system and  
7631 its software will be uncovered from time to time, but this is a byproduct of the primary activity,  
7632 debugging user code.7633 Another need for defining a trace interface in POSIX stems from the objective to provide an  
7634 efficient portable way to perform benchmarks. Existing practice shows that such interfaces are  
7635 commonly used in a variety of systems but with little commonality. As part of the benchmarking  
7636 needs, two aspects within the trace interface must be considered.

7637 The first, and perhaps more important one, is the qualitative aspect.

7638 The second is the quantitative aspect.

## 7639 • Qualitative Aspect

7640 To better understand this aspect, let us consider an example. Suppose that you want to  
7641 organize a number of actions to be performed during the day. Some of these actions are  
7642 known at the beginning of the day. Some others, which may be more or less important, will  
7643 be triggered by reading your mail. During the day you will make some phone calls and  
7644 synchronously receive some more information. Finally you will receive asynchronous phone  
7645 calls that also will trigger actions. If you, or somebody else, examines your day at work, you,  
7646 or he, can discover that you have not efficiently organized your work. For instance, relative  
7647 to the phone calls you made, would it be preferable to make some of these early in the  
7648 morning? Or to delay some others until the end of the day? Relative to the phone calls you  
7649 have received, you might find that somebody you called in the morning has called you 10  
7650 times while you were performing some important work. To examine, afterwards, your day at  
7651 work, you record in sequence all the trace events relative to your work. This should give you  
7652 a chance of organizing your next day at work.7653 This is the qualitative aspect of the trace interface. The user of a system needs to keep a trace  
7654 of particular points the application passes through, so that he can eventually make some  
7655 changes in the application and/or system configuration, to give the application a chance of  
7656 running more efficiently.

- Quantitative Aspect

This aspect concerns primarily realtime applications, where missed deadlines can be undesirable. Although there are, in IEEE Std 1003.1-2001, some interfaces useful for such applications (timeouts, execution time monitoring, and so on), there are no APIs to aid in the tuning of a realtime application's behavior (**timespec** in timeouts, length of message queues, duration of driver interrupt service routine, and so on). The tuning of an application needs a means of recording timestamped important trace events during execution in order to analyze offline, and eventually, to tune some realtime features (redesign the system with less functionalities, readjust timeouts, redesign driver interrupts, and so on).

### Detailed Objectives

Objectives were defined to build the trace interface and are kept for historical interest. Although some objectives are not fully respected in this trace interface, the concept of the POSIX trace interface assumes the following points:

1. It must be possible to trace both system and user trace events concurrently.
2. It must be possible to trace per-process trace events and also to trace system trace events which are unrelated to any particular process. A per-process trace event is either user-initiated or system-initiated.
3. It must be possible to control tracing on a per-process basis from either inside or outside the process.
4. It must be possible to control tracing on a per-thread basis from inside the enclosing process.
5. Trace points must be controllable by trace event type ID from inside and outside of the process. Multiple trace points can have the same trace event type ID, and will be controlled jointly.
6. Recording of trace events is dependent on both trace event type ID and the process/thread. Both must be enabled in order to record trace events. System trace events may or may not be handled differently.
7. The API must not mandate the ability to control tracing for more than one process at the same time.
8. There is no objective for trace control on anything bigger than a process; for example, group or session.
9. Trace propagation and control:
  - a. Trace propagation across *fork()* is optional; the default is to not trace a child process.
  - b. Trace control must span *pthread\_create()* operations; that is, if a process is being traced, any thread will be traced as well if this thread allows tracing. The default is to allow tracing.
10. Trace control must not span *exec* or *posix\_spawn()* operations.
11. A triggering API is not required. The triggering API is the ability to command or stop tracing based on the occurrence of a specific trace event other than a `POSIX_TRACE_START` trace event or a `POSIX_TRACE_STOP` trace event.
12. Trace log entries must have timestamps of implementation-defined resolution. Implementations are exhorted to support at least microsecond resolution. When a trace log entry is retrieved, it must have timestamp, PC address, PID, and TID of the entity that

- 7700 generated the trace event.
- 7701 13. Independently developed code should be able to use trace facilities without coordination  
7702 and without conflict.
- 7703 14. Even if the trace points in the trace calls are not unique, the trace log entries (after any  
7704 processing) must be uniquely identified as to trace point.
- 7705 15. There must be a standard API to read the trace stream.
- 7706 16. The format of the trace stream and the trace log is opaque and unspecified.
- 7707 17. It must be possible to read a completed trace, if recorded on some suitable non-volatile  
7708 storage, even subsequent to a power cycle or subsequent cold boot of the system.
- 7709 18. Support of analysis of a trace log while it is being formed is implementation-defined.
- 7710 19. The API must allow the application to write trace stream identification information into  
7711 the trace stream and to be able to retrieve it, without it being overwritten by trace entries,  
7712 even if the trace stream is full.
- 7713 20. It must be possible to specify the destination of trace data produced by trace events.
- 7714 21. It must be possible to have different trace streams, and for the tracing enabled by one trace  
7715 stream to be completely independent of the tracing of another trace stream.
- 7716 22. It must be possible to trace events from threads in different CPUs.
- 7717 23. The API must support one or more trace streams per-system, and one or more trace  
7718 streams per-process, up to an implementation-defined set of per-system and per-process  
7719 maximums.
- 7720 24. It must be possible to determine the order in which the trace events happened, without  
7721 necessarily depending on the clock, up to an implementation-defined time resolution.
- 7722 25. For performance reasons, the trace event point call(s) must be implementable as a macro  
7723 (see the ISO POSIX-1: 1996 standard, 1.3.4, Statement 2).
- 7724 26. IEEE Std 1003.1-2001 must not define the trace points which a conforming system must  
7725 implement, except for trace points used in the control of tracing.
- 7726 27. The APIs must be thread-safe, and trace points should be lock-free (that is, not require a  
7727 lock to gain exclusive access to some resource).
- 7728 28. The user-provided information associated with a trace event is variable-sized, up to some  
7729 maximum size.
- 7730 29. Bounds on record and trace stream sizes:
- 7731 a. The API must permit the application to declare the upper bounds on the length of an  
7732 application data record. The system must return the limit it used. The limit used may  
7733 be smaller than requested.
- 7734 b. The API must permit the application to declare the upper bounds on the size of trace  
7735 streams. The system must return the limit it used. The limit used may be different,  
7736 either larger or smaller, than requested.
- 7737 30. The API must be able to pass any fundamental data type, and a structured data type  
7738 composed only of fundamental types. The API must be able to pass data by reference,  
7739 given only as an address and a length. Fundamental types are the POSIX.1 types (see the  
7740 `<sys/types.h>` header) plus those defined in the ISO C standard.

31. The API must apply the POSIX notions of ownership and permission to recorded trace data, corresponding to the sources of that data.

### Comments on Objectives

**Note:** In the following comments, numbers in square brackets refer to the above objectives.

It is necessary to be able to obtain a trace stream for a complete activity. Thus there is a requirement to be able to trace both application and system trace events. A per-process trace event is either user-initiated, like the *write()* function, or system-initiated, like a timer expiration. There is also a need to be able to trace an entire process' activity even when it has threads in multiple CPUs. To avoid excess trace activity, it is necessary to be able to control tracing on a trace event type basis.

[Objectives 1,2,5,22]

There is a need to be able to control tracing on a per-process basis, both from inside and outside the process; that is, a process can start a trace activity on itself or any other process. There is also the perceived need to allow the definition of a maximum number of trace streams per system.

[Objectives 3,23]

From within a process, it is necessary to be able to control tracing on a per-thread basis. This provides an additional filtering capability to keep the amount of traced data to a minimum. It also allows for less ambiguity as to the origin of trace events. It is recognized that thread-level control is only valid from within the process itself. It is also desirable to know the maximum number of trace streams per process that can be started. The API should not require thread synchronization or mandate priority inversions that would cause the thread to block. However, the API must be thread-safe.

[Objectives 4,23,24,27]

There was no perceived objective to control tracing on anything larger than a process; for example, a group or session. Also, the ability to start or stop a trace activity on multiple processes atomically may be very difficult or cumbersome in some implementations.

[Objectives 6,8]

It is also necessary to be able to control tracing by trace event type identifier, sometimes called a trace hook ID. However, there is no mandated set of system trace events, since such trace points are implementation-defined. The API must not require from the operating system facilities that are not standard.

[Objectives 6,26]

Trace control must span *fork()* and *pthread\_create()*. If not, there will be no way to ensure that an application's activity is entirely traced. The newly forked child would not be able to turn on its tracing until after it obtained control after the fork, and trace control externally would be even more problematic.

[Objective 9]

Since *exec* and *posix\_spawn()* represent a complete change in the execution of a task (a new program), trace control need not persist over an *exec* or *posix\_spawn()*.

[Objective 10]

Where trace activities are started on multiple processes, these trace activities should not interfere with each other.

[Objective 21]

There is no need for a triggering objective, primarily for performance reasons; see also Section B.2.11.8 (on page 204), rationale on triggering.

[Objective 11]

7787 It must be possible to determine the origin of each traced event. The process and thread  
7788 identifiers for each trace event are needed. Also there was a perceived need for a user-specifiable  
7789 origin, but it was felt that this would create too much overhead.  
7790 [Objectives 12,14]

7791 An allowance must be made for trace points to come embedded in software components from  
7792 several different sources and vendors without requiring coordination.  
7793 [Objective 13]

7794 There is a requirement to be able to uniquely identify trace points that may have the same trace  
7795 stream identifier. This is only necessary when a trace report is produced.  
7796 [Objectives 12,14]

7797 Tracing is a very performance-sensitive activity, and will therefore likely be implemented at a  
7798 low level within the system. Hence the interface must not mandate any particular buffering or  
7799 storage method. Therefore, a standard API is needed to read a trace stream. Also the interface  
7800 must not mandate the format of the trace data, and the interface must not assume a trace storage  
7801 method. Due to the possibility of a monolithic kernel and the possible presence of multiple  
7802 processes capable of running trace activities, the two kinds of trace events may be stored in two  
7803 separate streams for performance reasons. A mandatory dump mechanism, common in some  
7804 existing practice, has been avoided to allow the implementation of this set of functions on small  
7805 realtime profiles for which the concept of a file system is not defined. The trace API calls should  
7806 be implemented as macros.  
7807 [Objectives 15,16,25,30]

7808 Since a trace facility is a valuable service tool, the output (or log) of a completed trace stream  
7809 that is written to permanent storage must be readable on other systems of the type that  
7810 produced the trace log. Note that there is no objective to be able to interpret a trace log that was  
7811 not successfully completed.  
7812 [Objectives 17,18,19]

7813 For trace streams written to permanent storage, a way to specify the destination of the trace  
7814 stream is needed.  
7815 [Objective 20]

7816 There is a requirement to be able to depend on the ordering of trace events up to some  
7817 implementation-defined time interval. For example, there is a need to know the time period  
7818 during which, if trace events are closer together, their ordering is unspecified. Events that occur  
7819 within an interval smaller than this resolution may or may not be read back in the correct order.  
7820 [Objective 24]

7821 The application should be able to know how much data can be traced. When trace event types  
7822 can be filtered, the application should be able to specify the approximate maximum amount of  
7823 data that will be traced in a trace event so resources can be more efficiently allocated.  
7824 [Objectives 28,29]

7825 Users should not be able to trace data to which they would not normally have access. System  
7826 trace events corresponding to a process/thread should be associated with the ownership of that  
7827 process/thread.  
7828 [Objective 31]

## 7829 B.2.11.2 Trace Model

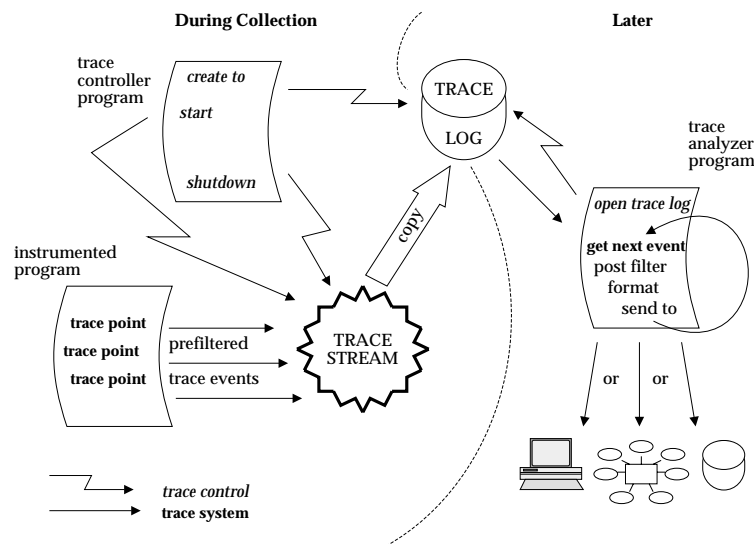
## 7830 Introduction

7831 The model is based on two base entities: the “Trace Stream” and the “Trace Log”, and a recorded  
 7832 unit called the “Trace Event”. The possibility of using Trace Streams and Trace Logs separately  
 7833 gives two use dimensions and solves both the performance issue and the full-information  
 7834 system issue. In the case of a trace stream without log, specific information, although reduced in  
 7835 quantity, is required to be registered, in a possibly small realtime system, with as little overhead  
 7836 as possible. The Trace Log option has been added for small realtime systems. In the case of a  
 7837 trace stream with log, considerable complex application-specific information needs to be  
 7838 collected.

## 7839 Trace Model Description

7840 The trace model can be examined for three different subfunctions: Application Instrumentation,  
 7841 Trace Operation Control, and Trace Analysis.

7842



7843

**Figure B-2** Trace System Overview: for Offline Analysis

7844 Each of these subfunctions requires specific characteristics of the trace mechanism API.

## 7845 • Application Instrumentation

7846 When instrumenting an application, the programmer is not concerned about the future use of  
 7847 the trace events in the trace stream or the trace log, the full policy of the trace stream, or the  
 7848 eventual pre-filtering of trace events. But he is concerned about the correct determination of  
 7849 the specific trace event type identifier, regardless of how many independent libraries are  
 7850 used in the same user application; see Figure B-2 and Figure B-3 (on page 187).

7851 This trace API provides the necessary operations to accomplish this subfunction. This is done  
 7852 by providing functions to associate a programmer-defined name with an implementation-  
 7853 defined trace event type identifier (see the `posix_trace_eventid_open()` function), and to send  
 7854 this trace event into a potential trace stream (see the `posix_trace_event()` function).

- Trace Operation Control

When controlling the recording of trace events in a trace stream, the programmer is concerned with the correct initialization of the trace mechanism (that is, the sizing of the trace stream), the correct retention of trace events in a permanent storage, the correct dynamic recording of trace events, and so on.

This trace API provides the necessary material to permit this efficiently. This is done by providing functions to initialize a new trace stream, and optionally a trace log:

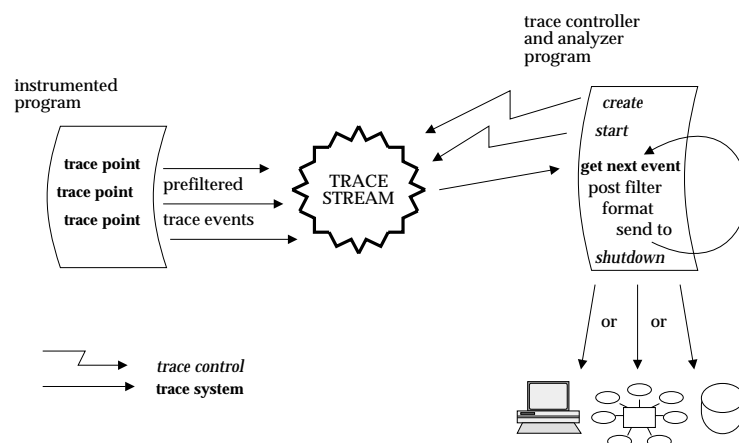
- Trace Stream Attributes Object Initialization (see *posix\_trace\_attr\_init()*)
- Functions to Retrieve or Set Information About a Trace Stream (see *posix\_trace\_attr\_getgenversion()*)
- Functions to Retrieve or Set the Behavior of a Trace Stream (see *posix\_trace\_attr\_getinherited()*)
- Functions to Retrieve or Set Trace Stream Size Attributes (see *posix\_trace\_attr\_getmaxusereventsize()*)
- Trace Stream Initialization, Flush, and Shutdown from a Process (see *posix\_trace\_create()*)
- Clear Trace Stream and Trace Log (see *posix\_trace\_clear()*)

To select the trace event types that are to be traced:

- Manipulate Trace Event Type Identifier (see *posix\_trace\_trid\_eventid\_open()*)
- Iterate over a Mapping of Trace Event Type (see *posix\_trace\_eventtypelist\_getnext\_id()*)
- Manipulate Trace Event Type Sets (see *posix\_trace\_eventset\_empty()*)
- Set Filter of an Initialized Trace Stream (see *posix\_trace\_set\_filter()*)

To control the execution of an active trace stream:

- Trace Start and Stop (see *posix\_trace\_start()*)
- Functions to Retrieve the Trace Attributes or Trace Statuses (see *posix\_trace\_get\_attr()*)



**Figure B-3** Trace System Overview: for Online Analysis

• Trace Analysis

Once correctly recorded, on permanent storage or not, an ultimate activity consists of the analysis of the recorded information. If the recorded data is on permanent storage, a specific open operation is required to associate a trace stream to a trace log.

The first intent of the group was to request the presence of a system identification structure in the trace stream attribute. This was, for the application, to allow some portable way to process the recorded information. However, there is no requirement that the **utsname** structure, on which this system identification was based, be portable from one machine to another, so the contents of the attribute cannot be interpreted correctly by an application conforming to IEEE Std 1003.1-2001.

This modification has been incorporated and requests that some unspecified information be recorded in the trace log in order to fail opening it if the analysis process and the controller process were running in different types of machine, but does not request that this information be accessible to the application. This modification has implied a modification in the *posix\_trace\_open()* function error code returns.

This trace API provides functions to:

- Extract trace stream identification attributes (see *posix\_trace\_attr\_getgenversion()*)
- Extract trace stream behavior attributes (see *posix\_trace\_attr\_getinherited()*)
- Extract trace event, stream, and log size attributes (see *posix\_trace\_attr\_getmaxuseventsizesize()*)
- Look up trace event type names (see *posix\_trace\_eventid\_get\_name()*)
- Iterate over trace event type identifiers (see *posix\_trace\_eventtypelist\_getnext\_id()*)
- Open, rewind, and close a trace log (see *posix\_trace\_open()*)
- Read trace stream attributes and status (see *posix\_trace\_get\_attr()*)
- Read trace events (see *posix\_trace\_getnext\_event()*)

Due to the following two reasons:

1. The requirement that the trace system must not add unacceptable overhead to the traced process and so that the trace event point execution must be fast
2. The traced application does not care about tracing errors

the trace system cannot return any internal error to the application. Internal error conditions can range from unrecoverable errors that will force the active trace stream to abort, to small errors that can affect the quality of tracing without aborting the trace stream. The group decided to define a system trace event to report to the analysis process such internal errors. It is not the intention of IEEE Std 1003.1-2001 to require an implementation to report an internal error that corrupts or terminates tracing operation. The implementor is free to decide which internal documented errors, if any, the trace system is able to report.



## States of a Trace Stream

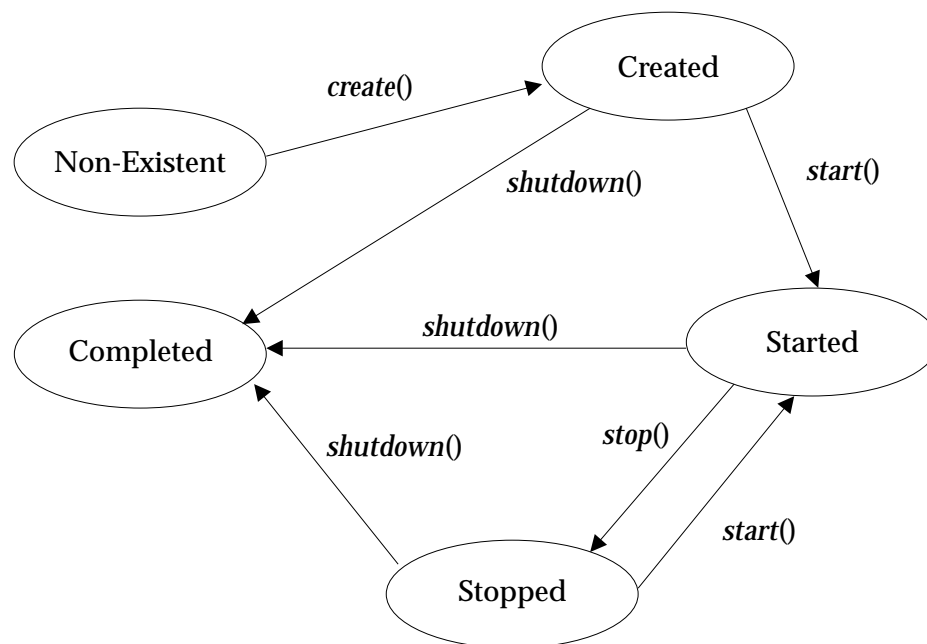


Figure B-4 Trace System Overview: States of a Trace Stream

Figure B-4 shows the different states an active trace stream passes through. After the `posix_trace_create()` function call, a trace stream becomes `CREATED` and a trace stream is associated for the future collection of trace events. The status of the trace stream is `POSIX_TRACE_SUSPENDED`. The state becomes `STARTED` after a call to the `posix_trace_start()` function, and the status becomes `POSIX_TRACE_RUNNING`. In this state, all trace events that are not filtered out will be stored into the trace stream. After a call to `posix_trace_stop()`, the trace stream becomes `STOPPED` (and the status `POSIX_TRACE_SUSPENDED`). In this state, no new trace events will be recorded in the trace stream, but previously recorded trace events may continue to be read.

After a call to `posix_trace_shutdown()`, the trace stream is in the state `COMPLETED`. The trace stream no longer exists but, if the Trace Log option is supported, all the information contained in it has been logged. If a log object has not been associated with the trace stream at the creation, it is the responsibility of the trace controller process to not shut the trace stream down while trace events remain to be read in the stream.

## Tracing All Processes

Some implementations have a tracing subsystem with the ability to trace all processes. This is useful to debug some types of device drivers such as those for ATM or X25 adapters. These types of adapters are used by several independent processes, that are not issued from the same process.

The POSIX trace interface does not define any constant or option to create a trace stream tracing all processes. POSIX.1 does not prevent this type of implementation and an implementor is free to add this capability. Nevertheless, the trace interface allows tracing of all the system trace events and all the processes issued from the same process.

If such a tracing system capability has to be implemented, when a trace stream is created, it is recommended that a constant named `POSIX_TRACE_ALLPROC` be used instead of the process identifier in the argument of the `posix_trace_create()` or `posix_trace_create_withlog()` function. A possible value for `POSIX_TRACE_ALLPROC` may be `-1` instead of a real process identifier.

The implementor has to be aware that there is some impact on the tracing behavior as defined in the POSIX trace interface. For example:

- If the default value for the inheritance attribute is set to `POSIX_TRACE_CLOSE_FOR_CHILD`, the implementation has to stop tracing for the child process.
- The trace controller which is creating this type of trace stream must have the appropriate privilege to trace all the processes.

## Trace Storage

The model is based on two types of trace events: system trace events and user-defined trace events. The internal representation of trace events is implementation-defined, and so the implementor is free to choose the more suitable, practical, and efficient way to design the internal management of trace events. For the timestamping operation, the model does not impose the `CLOCK_REALTIME` or any other clock. The buffering allocation and operation follow the same principle. The implementor is free to use one or more buffers to record trace events; the interface assumes only a logical trace stream of sequentially recorded trace events. Regarding flushing of trace events, the interface allows the definition of a trace log object which typically can be a file. But the group was also aware of defining functions to permit the use of this interface in small realtime systems, which may not have general file system capabilities. For instance, the three functions `posix_trace_getnext_event()` (blocking), `posix_trace_timedgetnext_event()` (blocking with timeout), and `posix_trace_trygetnext_event()` (non-blocking) are proposed to read the recorded trace events.

The policy to be used when the trace stream becomes full also relies on common practice:

- For an active trace stream, the `POSIX_TRACE_LOOP` trace stream policy permits automatic overrun (overwrite of oldest trace events) while waiting for some user-defined condition to cause tracing to stop. By contrast, the `POSIX_TRACE_UNTIL_FULL` trace stream policy requires the system to stop tracing when the trace stream is full. However, if the trace stream that is full is at least partially emptied by a call to the `posix_trace_flush()` function or by calls to the `posix_trace_getnext_event()` function, the trace system will automatically resume tracing.

If the Trace Log option is supported, the operation of the `POSIX_TRACE_FLUSH` policy is an extension of the `POSIX_TRACE_UNTIL_FULL` policy. The automatic free operation (by flushing to the associated trace log) is added.

- If a log is associated with the trace stream and this log is a regular file, these policies also apply for the log. One more policy, `POSIX_TRACE_APPEND`, is defined to allow indefinite extension of the log. Since the log destination can be any device or pseudo-device, the implementation may not be able to manipulate the destination as required by IEEE Std 1003.1-2001. For this reason, the behavior of the log full policy may be unspecified depending on the trace log type.

The current trace interface does not define a service to preallocate space for a trace log file, because this space can be preallocated by means of a call to the `posix_fallocate()` function. This function could be called after the file has been opened, but before the trace stream is created. The `posix_fallocate()` function ensures that any required storage for regular file data is allocated on the file system storage media. If `posix_fallocate()` returns successfully,

7990 subsequent writes to the specified file data will not fail due to the lack of free space on the file  
 7991 system storage media. Besides trace events, a trace stream also includes trace attributes and  
 7992 the mapping from trace event names to trace event type identifiers. The implementor is free  
 7993 to choose how to store the trace attributes and the trace event type map, but must ensure that  
 7994 this information is not lost when a trace stream overrun occurs.

### 7995 *B.2.11.3 Trace Programming Examples*

7996 Several programming examples are presented to show the code of the different possible  
 7997 subfunctions using a trace subsystem. All these programs need to include the `<trace.h>` header.  
 7998 In the examples shown, error checking is omitted for more simplicity.

### 7999 **Trace Operation Control**

8000 These examples show the creation of a trace stream for another process; one which is already  
 8001 trace instrumented. All the default trace stream attributes are used to simplify programming in  
 8002 the first example. The second example shows more possibilities.

### 8003 **First Example**

```

8004 /* Caution. Error checks omitted */
8005 {
8006     trace_attr_t attr;
8007     pid_t pid = traced_process_pid;
8008     int fd;
8009     trace_id_t trid;
8010
8011     - - - - -
8012     /* Initialize trace stream attributes */
8013     posix_trace_attr_init(&attr);
8014     /* Open a trace log */
8015     fd=open("/tmp/mytracelog",...);
8016     /*
8017      * Create a new trace associated with a log
8018      * and with default attributes
8019      */
8020     posix_trace_create_withlog(pid, &attr, fd, &trid);
8021
8022     /* Trace attribute structure can now be destroyed */
8023     posix_trace_attr_destroy(&attr);
8024     /* Start of trace event recording */
8025     posix_trace_start(trid);
8026     - - - - -
8027     /* Duration of tracing */
8028     - - - - -
8029     /* Stop and shutdown of trace activity */
8030     posix_trace_shutdown(trid);
8031     - - - - -
8032 }
```

**Second Example**

Between the initialization of the trace stream attributes and the creation of the trace stream, these trace stream attributes may be modified; see **Trace Stream Attribute Manipulation** (on page 196) for a specific programming example. Between the creation and the start of the trace stream, the event filter may be set; after the trace stream is started, the event filter may be changed. The setting of an event set and the change of a filter is shown in **Create a Trace Event Type Set and Change the Trace Event Type Filter** (on page 196).

```

/* Caution. Error checks omitted */
{
    trace_attr_t attr;
    pid_t pid = traced_process_pid;
    int fd;
    trace_id_t trid;
    - - - - -
    /* Initialize trace stream attributes */
    posix_trace_attr_init(&attr);
    /* Attr default may be changed at this place; see example */
    - - - - -
    /* Create and open a trace log with R/W user access */
    fd=open("/tmp/mytracelog",O_WRONLY|O_CREAT,S_IRUSR|S_IWUSR);
    /* Create a new trace associated with a log */
    posix_trace_create_withlog(pid, &attr, fd, &trid);
    /*
     * If the Trace Filter option is supported
     * trace event type filter default may be changed at this place;
     * see example about changing the trace event type filter
     */
    posix_trace_start(trid);
    - - - - -

    /*
     * If you have an uninteresting part of the application
     * you can stop temporarily.
     *
     * posix_trace_stop(trid);
     * - - - - -
     * - - - - -
     * posix_trace_start(trid);
     */
    - - - - -

    /*
     * If the Trace Filter option is supported
     * the current trace event type filter can be changed
     * at any time (see example about how to set
     * a trace event type filter)
     */
    - - - - -

    /* Stop the recording of trace events */
    posix_trace_stop(trid);
    /* Shutdown the trace stream */
    posix_trace_shutdown(trid);

```

```

8083      /*
8084      * Destroy trace stream attributes; attr structure may have
8085      * been used during tracing to fetch the attributes
8086      */
8087      posix_trace_attr_destroy(&attr);
8088      - - - - -
8089  }

```

## 8090 Application Instrumentation

8091 This example shows an instrumented application. The code is included in a block of instructions,  
 8092 perhaps a function from a library. Possibly in an initialization part of the instrumented  
 8093 application, two user trace events names are mapped to two trace event type identifiers  
 8094 (function *posix\_trace\_eventid\_open()*). Then two trace points are programmed.

```

8095  /* Caution. Error checks omitted */
8096  {
8097      trace_event_id_t eventid1, eventid2;
8098      - - - - -
8099      /* Initialization of two trace event type ids */
8100      posix_trace_eventid_open("my_first_event",&eventid1);
8101      posix_trace_eventid_open("my_second_event",&eventid2);
8102      - - - - -
8103      - - - - -
8104      - - - - -
8105      /* Trace point */
8106      posix_trace_event(eventid1,NULL,0);
8107      - - - - -
8108      /* Trace point */
8109      posix_trace_event(eventid2,NULL,0);
8110      - - - - -
8111  }

```

## 8112 Trace Analyzer

8113 This example shows the manipulation of a trace log resulting from the dumping of a completed  
 8114 trace stream. All the default attributes are used to simplify programming, and data associated  
 8115 with a trace event is not shown in the first example. The second example shows more  
 8116 possibilities.

## 8117 First Example

```

8118  /* Caution. Error checks omitted */
8119  {
8120      int fd;
8121      trace_id_t trid;
8122      posix_trace_event_info trace_event;
8123      char trace_event_name[TRACE_EVENT_NAME_MAX];
8124      int return_value;
8125      size_t returndatasize;
8126      int lost_event_number;
8127      - - - - -

```

```

8128      /* Open an existing trace log */
8129      fd=open("/tmp/tracelog", O_RDONLY);
8130      /* Open a trace stream on the open log */
8131      posix_trace_open(fd, &trid);
8132      /* Read a trace event */
8133      posix_trace_getnext_event(trid, &trace_event,
8134          NULL, 0, &returndatasize,&return_value);

8135      /* Read and print all trace event names out in a loop */
8136      while (return_value == NULL)
8137      {
8138          /*
8139           * Get the name of the trace event associated
8140           * with trid trace ID
8141           */
8142          posix_trace_eventid_get_name(trid, trace_event.event_id,
8143              trace_event_name);
8144          /* Print the trace event name out */
8145          printf("%s\n",trace_event_name);
8146          /* Read a trace event */
8147          posix_trace_getnext_event(trid, &trace_event,
8148              NULL, 0, &returndatasize,&return_value);
8149      }

8150      /* Close the trace stream */
8151      posix_trace_close(trid);
8152      /* Close the trace log */
8153      close(fd);
8154  }

```

## 8155 Second Example

8156 The complete example includes the two other examples in **Retrieve Information from a Trace**  
8157 **Log** (on page 197) and in **Retrieve the List of Trace Event Types Used in a Trace Log** (on page  
8158 198). For example, the *maxdatasize* variable is set in **Retrieve the List of Trace Event Types Used**  
8159 **in a Trace Log** (on page 198).

```

8160      /* Caution. Error checks omitted */
8161      {
8162          int fd;
8163          trace_id_t trid;
8164          posix_trace_event_info trace_event;
8165          char trace_event_name[TRACE_EVENT_NAME_MAX];
8166          char * data;
8167          size_t maxdatasize=1024, returndatasize;
8168          int return_value;
8169          - - - - -

8170          /* Open an existing trace log */
8171          fd=open("/tmp/tracelog", O_RDONLY);
8172          /* Open a trace stream on the open log */
8173          posix_trace_open( fd, &trid);
8174          /*
8175           * Retrieve information about the trace stream which

```

```

8176      * was dumped in this trace log (see example)
8177      */
8178      - - - - -
8179      /* Allocate a buffer for trace event data */
8180      data=(char *)malloc(maxdatasize);
8181      /*
8182      * Retrieve the list of trace events used in this
8183      * trace log (see example)
8184      */
8185      - - - - -
8186      /* Read and print all trace event names and data out in a loop */
8187      while (1)
8188      {
8189      posix_trace_getnext_event(trid, &trace_event,
8190      data, maxdatasize, &returndatasize,&return_value);
8191      if (return_value != NULL) break;
8192      /*
8193      * Get the name of the trace event type associated
8194      * with trid trace ID
8195      */
8196      posix_trace_eventid_get_name(trid, trace_event.event_id,
8197      trace_event_name);
8198      {
8199      int i;
8200      /* Print the trace event name out */
8201      printf("%s: ", trace_event_name);
8202      /* Print the trace event data out */
8203      for (i=0; i<returndatasize, i++) printf("%02.2X",
8204      (unsigned char)data[i]);
8205      printf("\n");
8206      }
8207      }
8208      /* Close the trace stream */
8209      posix_trace_close(trid);
8210      /* The buffer data is deallocated */
8211      free(data);
8212      /* Now the file can be closed */
8213      close(fd);
8214      }

```

## 8215 **Several Programming Manipulations**

8216 The following examples show some typical sets of operations needed in some contexts.

**Trace Stream Attribute Manipulation**

This example shows the manipulation of a trace stream attribute object in order to change the default value provided by a previous *posix\_trace\_attr\_init()* call.

```

/* Caution. Error checks omitted */
{
    trace_attr_t attr;
    size_t logsize=100000;
    - - - - -
    /* Initialize trace stream attributes */
    posix_trace_attr_init(&attr);
    /* Set the trace name in the attributes structure */
    posix_trace_attr_setname(&attr, "my_trace");
    /* Set the trace full policy */
    posix_trace_attr_setstreamfullpolicy(&attr, POSIX_TRACE_LOOP);
    /* Set the trace log size */
    posix_trace_attr_setlogsize(&attr, logsize);
    - - - - -
}

```

**Create a Trace Event Type Set and Change the Trace Event Type Filter**

This example is valid only if the Trace Event Filter option is supported. This example shows the manipulation of a trace event type set in order to change the trace event type filter for an existing active trace stream, which may be just-created, running, or suspended. Some sets of trace event types are well-known, such as the set of trace event types not associated with a process, some trace event types are just-built trace event types for this trace stream; one trace event type is the predefined trace event error type which is deleted from the trace event type set.

```

/* Caution. Error checks omitted */
{
    trace_id_t trid = existing_trace;
    trace_event_set_t set;
    trace_event_id_t trace_event1, trace_event2;
    - - - - -
    /* Initialize to an empty set of trace event types */
    /* (not strictly required because posix_trace_event_set_fill() */
    /* will ignore the prior contents of the event set.) */
    posix_trace_eventset_emptyset(&set);
    /*
     * Fill the set with all system trace events
     * not associated with a process
     */
    posix_trace_eventset_fill(&set, POSIX_TRACE_WOPID_EVENTS);

    /*
     * Get the trace event type identifier of the known trace event name
     * my_first_event for the trid trace stream
     */
    posix_trace_trid_eventid_open(trid, "my_first_event", &trace_event1);
    /* Add the set with this trace event type identifier */
    posix_trace_eventset_add_event(trace_event1, &set);
    /*

```



```

8265      * Get the trace event type identifier of the known trace event name
8266      * my_second_event for the trid trace stream
8267      */
8268      posix_trace_trid_eventid_open(trid, "my_second_event", &trace_event2);
8269      /* Add the set with this trace event type identifier */
8270      posix_trace_eventset_add_event(trace_event2, &set);
8271      - - - - -
8272      /* Delete the system trace event POSIX_TRACE_ERROR from the set */
8273      posix_trace_eventset_del_event(POSIX_TRACE_ERROR, &set);
8274      - - - - -
8275      /* Modify the trace stream filter making it equal to the new set */
8276      posix_trace_set_filter(trid, &set, POSIX_TRACE_SET_EVENTSET);
8277      - - - - -
8278      /*
8279      * Now trace_event1, trace_event2, and all system trace event types
8280      * not associated with a process, except for the POSIX_TRACE_ERROR
8281      * system trace event type, are filtered out of (not recorded in) the
8282      * existing trace stream.
8283      */
8284  }

```

### Retrieve Information from a Trace Log

This example shows how to extract information from a trace log, the dump of a trace stream. This code:

- Asks if the trace stream has lost trace events
- Extracts the information about the version of the trace subsystem which generated this trace log
- Retrieves the maximum size of trace event data; this may be used to dynamically allocate an array for extracting trace event data from the trace log without overflow

```

8293  /* Caution. Error checks omitted */
8294  {
8295      struct posix_trace_status_info statusinfo;
8296      trace_attr_t attr;
8297      trace_id_t trid = existing_trace;
8298      size_t maxdatasize;
8299      char genversion[TRACE_NAME_MAX];
8300      - - - - -
8301      /* Get the trace stream status */
8302      posix_trace_get_status(trid, &statusinfo);
8303      /* Detect an overrun condition */
8304      if (statusinfo.posix_stream_overrun_status == POSIX_TRACE_OVERRUN)
8305          printf("trace events have been lost\n");
8306      /* Get attributes from the trid trace stream */
8307      posix_trace_get_attr(trid, &attr);
8308      /* Get the trace generation version from the attributes */
8309      posix_trace_attr_getgenversion(&attr, genversion);
8310      /* Print the trace generation version out */
8311      printf("Information about Trace Generator:%s\n", genversion);

```

```

8312      /* Get the trace event max data size from the attributes */
8313      posix_trace_attr_getmaxdatasize(&attr, &maxdatasize);
8314      /* Print the trace event max data size out */
8315      printf("Maximum size of associated data:%d\n",maxdatasize);
8316      /* Destroy the trace stream attributes */
8317      posix_trace_attr_destroy(&attr);
8318  }

```

### 8319 **Retrieve the List of Trace Event Types Used in a Trace Log**

8320 This example shows the retrieval of a trace stream's trace event type list. This operation may be  
 8321 very useful if you are interested only in tracking the type of trace events in a trace log.

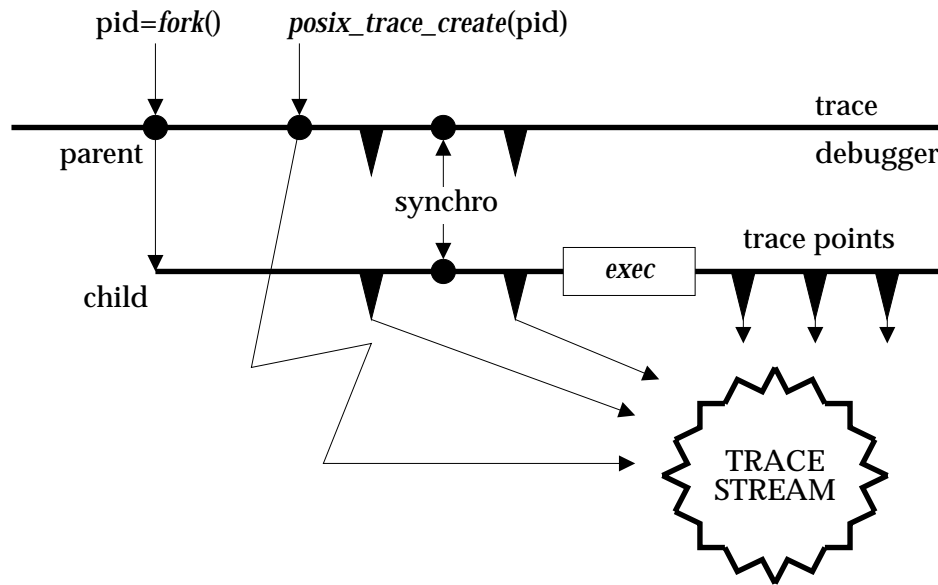
```

8322      /* Caution. Error checks omitted */
8323      {
8324          trace_id_t trid = existing_trace;
8325          trace_event_id_t event_id;
8326          char event_name[TRACE_EVENT_NAME_MAX];
8327          int return_value;
8328          - - - - -
8329          /*
8330           * In a loop print all existing trace event names out
8331           * for the trid trace stream
8332           */
8333          while (1)
8334          {
8335              posix_trace_eventtypelist_getnext_id(trid, &event_id
8336              &return_value);
8337              if (return_value != NULL) break;
8338              /*
8339               * Get the name of the trace event associated
8340               * with trid trace ID
8341               */
8342              posix_trace_eventid_get_name(trid, event_id, event_name);
8343              /* Print the name out */
8344              printf("%s\n", event_name);
8345          }
8346      }

```

## 8347 B.2.11.4 Rationale on Trace for Debugging

8348



8349

Figure B-5 Trace Another Process

8350

8351

8352

8353

8354

8355

8356

Among the different possibilities offered by the trace interface defined in IEEE Std 1003.1-2001, the debugging of an application is the most interesting one. Typical operations in the controlling debugger process are to filter trace event types, to get trace events from the trace stream, to stop the trace stream when the debugged process is executing uninteresting code, to start the trace stream when some interesting point is reached, and so on. The interface defined in IEEE Std 1003.1-2001 should define all the necessary base functions to allow this dynamic debug handling.

8357

8358

8359

8360

Figure B-5 shows an example in which the trace stream is created after the call to the `fork()` function. If the user does not want to lose trace events, some synchronization mechanism (represented in the figure) may be needed before calling the `exec` function, to give the parent a chance to create the trace stream before the child begins the execution of its trace points.

## 8361 B.2.11.5 Rationale on Trace Event Type Name Space

8362

8363

8364

8365

8366

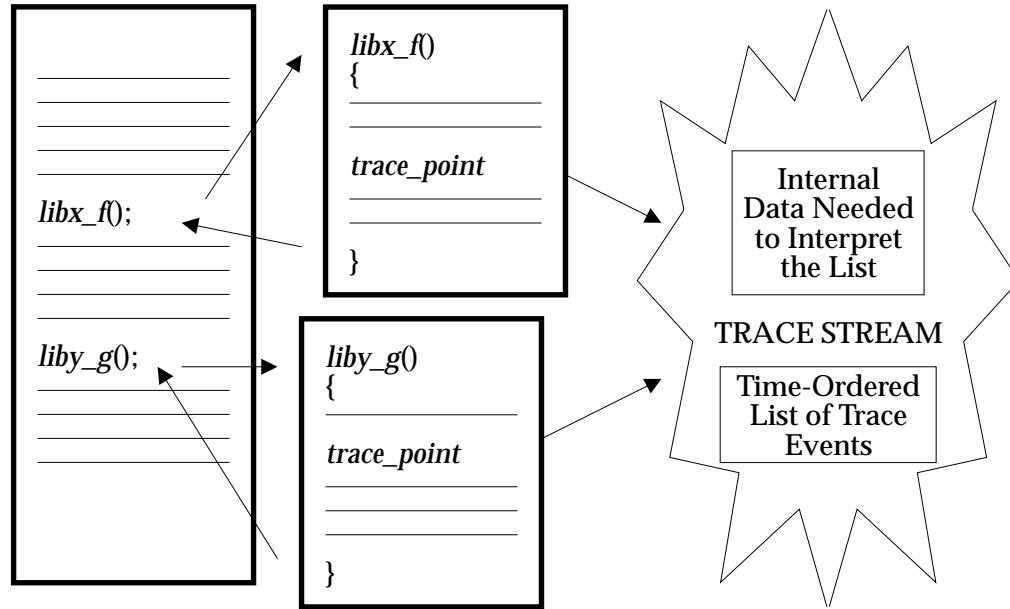
8367

8368

8369

At first, the working group was in favor of the representation of a trace event type by an integer (`event_name`). It seems that existing practice shows the weakness of such a representation. The collision of trace event types is the main problem that cannot be simply resolved using this sort of representation. Suppose, for example, that a third party designs an instrumented library. The user does not have the source of this library and wants to trace his application which uses in some part the third-party library. There is no means for him to know what are the trace event types used in the instrumented library so he has some chance of duplicating some of them and thus to obtain a contaminated tracing of his application.

8370



8371

**Figure B-6** Trace Name Space Overview: With Third-Party Library

8372

8373

8374

8375

8376

8377

8378

8379

8380

8381

8382

There are requirements to allow program images containing pieces from various vendors to be traced without also requiring those of any other vendors to coordinate their uses of the trace facility, and especially the naming of their various trace event types and trace point IDs. The chosen solution is to provide a very large name space, large enough so that the individual vendors can give their trace types and tracepoint IDs sufficiently long and descriptive names making the occurrence of collisions quite unlikely. The probability of collision is thus made sufficiently low so that the problem may, as a practical matter, be ignored. By requirement, the consequence of collisions will be a slight ambiguity in the trace streams; tracing will continue in spite of collisions and ambiguities. “The show must go on”. The *posix\_prog\_address* member of the **posix\_trace\_event\_info** structure is used to allow trace streams to be unambiguously interpreted, despite the fact that trace event types and trace event names need not be unique.

8383

8384

8385

8386

8387

8388

8389

8390

8391

8392

The *posix\_trace\_eventid\_open()* function is required to allow the instrumented third-party library to get a valid trace event type identifier for its trace event names. This operation is, somehow, an allocation, and the group was aware of proposing some deallocation mechanism which the instrumented application could use to recover the resources used by a trace event type identifier. This would have given the instrumented application the benefit of being capable of reusing a possible minimum set of trace event type identifiers, but also the inconvenience to have, possibly in the same trace stream, one trace event type identifier identifying two different trace event types. After some discussions the group decided to not define such a function which would make this API thicker for little benefit, the user having always the possibility of adding identification information in the *data* member of the trace event structure.

8393

8394

8395

8396

8397

8398

8399

The set of the trace event type identifiers the controlling process wants to filter out is initialized in the trace mechanism using the function *posix\_trace\_set\_filter()*, setting the arguments according to the definitions explained in *posix\_trace\_set\_filter()*. This operation can be done statically (when the trace is in the STOPPED state) or dynamically (when the trace is in the STARTED state). The preparation of the filter is normally done using the function defined in *posix\_trace\_eventtypelist\_getnext\_id()* and eventually the function *posix\_trace\_eventtypelist\_rewind()* in order to know (before the recording) the list of the potential

set of trace event types that can be recorded. In the case of an active trace stream, this list may not be exhaustive. Actually, the target process may not have yet called the function `posix_trace_eventid_open()`. But it is a common practice, for a controlling process, to prepare the filtering of a future trace stream before its start. Therefore the user must have a way to get the trace event type identifier corresponding to a well-known trace event name before its future association by the pre-cited function. This is done by calling the `posix_trace_trid_eventid_open()` function, given the trace stream identifier and the trace name, and described hereafter. Because this trace event type identifier is associated with a trace stream identifier, where a unique process has initialized two or more traces, the implementation is expected to return the same trace event type identifier for successive calls to `posix_trace_trid_eventid_open()` with different trace stream identifiers. The `posix_trace_eventid_get_name()` function is used by the controller process to identify, by the name, the trace event type returned by a call to the `posix_trace_eventtypelist_getnext_id()` function.

Afterwards, the set of trace event types is constructed using the functions defined in `posix_trace_eventset_empty()`, `posix_trace_eventset_fill()`, `posix_trace_eventset_add()`, and `posix_trace_eventset_del()`.

A set of functions is provided devoted to the manipulation of the trace event type identifier and names for an active trace stream. All these functions require the trace stream identifier argument as the first parameter. The opacity of the trace event type identifier implies that the user cannot associate directly its well-known trace event name with the system-associated trace event type identifier.

The `posix_trace_trid_eventid_open()` function allows the application to get the system trace event type identifier back from the system, given its well-known trace event name. This function is useful only when a controlling process needs to specify specific events to be filtered.

The `posix_trace_eventid_get_name()` function allows the application to obtain a trace event name given its trace event type identifier. One possible use of this function is to identify the type of a trace event retrieved from the trace stream, and print it. The easiest way to implement this requirement, is to use a single trace event type map for all the processes whose maps are required to be identical. A more difficult way is to attempt to keep multiple maps identical at every call to `posix_trace_eventid_open()` and `posix_trace_trid_eventid_open()`.

#### B.2.11.6 Rationale on Trace Events Type Filtering

The most basic rationale for runtime and pre-registration filtering (selection/rejection) of trace event types is to prevent choking of the trace collection facility, and/or overloading of the computer system. Any worthwhile trace facility can bring even the largest computer to its knees. Otherwise, everything would be recorded and filtered after the fact; it would be much simpler, but impractical.

To achieve debugging, measurement, or whatever the purpose of tracing, the filtering of trace event types is an important part of trace analysis. Due to the fact that the trace events are put into a trace stream and probably logged afterwards into a file, different levels of filtering—that is, rejection of trace event types—are possible.

**Filtering of Trace Event Types Before Tracing**

This function, represented by the `posix_trace_set_filter()` function in IEEE Std 1003.1-2001 (see `posix_trace_set_filter()`), selects, before or during tracing, the set of trace event types to be filtered out. It should be possible also (as OSF suggested in their ETAP trace specifications) to select the kernel trace event types to be traced in a system-wide fashion. These two functionalities are called the pre-filtering of trace event types.

The restriction on the actual type used for the `trace_event_set_t` type is intended to guarantee that these objects can always be assigned, have their address taken, and be passed by value as parameters. It is not intended that this type be a structure including pointers to other data structures, as that could impact the portability of applications performing such operations. A reasonable implementation could be a structure containing an array of integer types.

**Filtering of Trace Event Types at Runtime**

It is possible to build this functionality using the `posix_trace_set_filter()` function. A privileged process or a privileged thread can get trace events from the trace stream of another process or thread, and thus specify the type of trace events to record into a file, using implementation-defined methods and interfaces. This functionality, called inline filtering of trace event types, is used for runtime analysis of trace streams.

**Post-Mortem Filtering of Trace Event Types**

The word “post-mortem” is used here to indicate that some unanticipated situation occurs during execution that does not permit a pre or inline filtering of trace events and that it is necessary to record all trace event types to have a chance to discover the problem afterwards. When the program stops, all the trace events recorded previously can be analyzed in order to find the solution. This functionality could be named the post-filtering of trace event types.

**Discussions about Trace Event Type-Filtering**

After long discussions with the parties involved in the process of defining the trace interface, it seems that the sensitivity to the filtering problem is different, but everybody agrees that the level of the overhead introduced during the tracing operation depends on the filtering method elected. If the time that it takes the trace event to be recorded can be neglected, the overhead introduced by the filtering process can be classified as follows:

Pre-filtering	System and process/thread-level overhead
Inline-filtering	Process/thread-level overhead
Post-filtering	No overhead; done offline

The pre-filtering could be named “critical realtime” filtering in the sense that the filtering of trace event type is manageable at the user level so the user can lower to a minimum the filtering overhead at some user selected level of priority for the inline filtering, or delay the filtering to after execution for the post-filtering. The counterpart of this solution is that the size of the trace stream must be sufficient to record all the trace events. The advantage of the pre-filtering is that the utilization of the trace stream is optimized.

Only pre-filtering is defined by IEEE Std 1003.1-2001. However, great care must be taken in specifying pre-filtering, so that it does not impose unacceptable overhead. Moreover, it is necessary to isolate all the functionality relative to the pre-filtering.

The result of this rationale is to define a new option, the Trace Event Filter option, not necessarily implemented in small realtime systems, where system overhead is minimized to the extent possible.

8484 B.2.11.7 Tracing, *pthread* API

8485 The objective to be able to control tracing for individual threads may be in conflict with the  
 8486 efficiency expected in threads with a *contentionscope* attribute of `PTHREAD_SCOPE_PROCESS`.  
 8487 For these threads, context switches from one thread that has tracing enabled to another thread  
 8488 that has tracing disabled may require a kernel call to inform the kernel whether it has to trace  
 8489 system events executed by that thread or not. For this reason, it was proposed that the ability to  
 8490 enable or disable tracing for `PTHREAD_SCOPE_PROCESS` threads be made optional, through  
 8491 the introduction of a Trace Scope Process option. A trace implementation which did not  
 8492 implement the Trace Scope Process option would not honor the tracing-state attribute of a  
 8493 thread with `PTHREAD_SCOPE_PROCESS`; it would, however, honor the tracing-state attribute  
 8494 of a thread with `PTHREAD_SCOPE_SYSTEM`. This proposal was rejected as:

- 8495 1. Removing desired functionality (per-thread trace control)
- 8496 2. Introducing counter-intuitive behavior for the tracing-state attribute
- 8497 3. Mixing logically orthogonal ideas (thread scheduling and thread tracing)
- 8498 [Objective 4]

8499 Finally, to solve this complex issue, this API does not provide *pthread\_gettracingstate()*,  
 8500 *pthread\_settracingstate()*, *pthread\_attr\_gettracingstate()*, and *pthread\_attr\_settracingstate()*  
 8501 interfaces. These interfaces force the thread implementation to add to the weight of the thread  
 8502 and cause a revision of the threads libraries, just to support tracing. Worse yet,  
 8503 *posix\_trace\_event()* must always test this per-thread variable even in the common case where it is  
 8504 not used at all. Per-thread tracing is easy to implement using existing interfaces where necessary;  
 8505 see the following example.

8506 **Example**

```
8507 /* Caution. Error checks omitted */
8508 static pthread_key_t my_key;
8509 static trace_event_id_t my_event_id;
8510 static pthread_once_t my_once = PTHREAD_ONCE_INIT;

8511 void my_init(void)
8512 {
8513     (void) pthread_key_create(&my_key, NULL);
8514     (void) posix_trace_eventid_open("my", &my_event_id);
8515 }

8516 int get_trace_flag(void)
8517 {
8518     pthread_once(&my_once, my_init);
8519     return (pthread_getspecific(my_key) != NULL);
8520 }

8521 void set_trace_flag(int f)
8522 {
8523     pthread_once(&my_once, my_init);
8524     pthread_setspecific(my_key, f? &my_event_id: NULL);
8525 }

8526 fn( )
8527 {
8528     if (get_trace_flag())
8529         posix_trace_event(my_event_id, ...)
```

8530 }

8531 The above example does not implement third-party state setting.

8532 Lastly, per-thread tracing works poorly for threads with PTHREAD\_SCOPE\_PROCESS  
8533 contention scope. These “library” threads have minimal interaction with the kernel and would  
8534 have to explicitly set the attributes whenever they are context switched to a new kernel thread in  
8535 order to trace system events. Such state was explicitly avoided in POSIX threads to keep  
8536 PTHREAD\_SCOPE\_PROCESS threads lightweight.

8537 The reason that keeping PTHREAD\_SCOPE\_PROCESS threads lightweight is important is that  
8538 such threads can be used not just for simple multi-processors but also for co-routine style  
8539 programming (such as discrete event simulation) without inventing a new threads paradigm.  
8540 Adding extra runtime cost to thread context switches will make using POSIX threads less  
8541 attractive in these situations.

#### 8542 *B.2.11.8 Rationale on Triggering*

8543 The ability to start or stop tracing based on the occurrence of specific trace event types has been  
8544 proposed as a parallel to similar functionality appearing in logic analyzers. Such triggering, in  
8545 order to be very useful, should be based not only on the trace event type, but on trace event-  
8546 specific data, including tests of user-specified fields for matching or threshold values.

8547 Such a facility is unnecessary where the buffering of the stream is not a constraint, since such  
8548 checks can be performed offline during post-mortem analysis.

8549 For example, a large system could incorporate a daemon utility to collect the trace records from  
8550 memory buffers and spool them to secondary storage for later analysis. In the instances where  
8551 resources are truly limited, such as embedded applications, the application incorporation of  
8552 application code to test the circumstances of a trace event and call the trace point only if needed  
8553 is usually straightforward.

8554 For performance reasons, the *posix\_trace\_event()* function should be implemented using a macro,  
8555 so if the trace is inactive, the trace event point calls are latent code and must cost no more than a  
8556 scalar test.

8557 The API proposed in IEEE Std 1003.1-2001 does not include any triggering functionality.

#### 8558 *B.2.11.9 Rationale on Timestamp Clock*

8559 It has been suggested that the tracing mechanism should include the possibility of specifying the  
8560 clock to be used in timestamping the trace events. When application trace events must be  
8561 correlated to remote trace events, such a facility could provide a global time reference not  
8562 available from a local clock. Further, the application may be driven by timers based on a clock  
8563 different from that used for the timestamp, and the correlation of the trace to those untraced  
8564 timer activities could be an important part of the analysis of the application.

8565 However, the tracing mechanism needs to be fast and just the provision of such an option can  
8566 materially affect its performance. Leaving aside the performance costs of reading some clocks,  
8567 this notion is also ill-defined when kernel trace events are to be traced by two applications  
8568 making use of different tracing clocks. This can even happen within a single application where  
8569 different parts of the application are served by different clocks. Another complication can occur  
8570 when a clock is maintained strictly at the user level and is unavailable at the kernel level.

8571 It is felt that the benefits of a selectable trace clock do not match its costs. Applications that wish  
8572 to correlate clocks other than the default tracing clock can include trace events with sample  
8573 values of those other clocks, allowing correlation of timestamps from the various independent  
8574 clocks. In any case, such a technique would be required when applications are sensitive to



8575 multiple clocks.

#### 8576 B.2.11.10 Rationale on Different Overrun Conditions

8577 The analysis of the dynamic behavior of the trace mechanism shows that different overrun  
8578 conditions may occur. The API must provide a means to manage such conditions in a portable  
8579 way.

#### 8580 **Overrun in Trace Streams Initialized with POSIX\_TRACE\_LOOP Policy**

8581 In this case, the user of the trace mechanism is interested in using the trace stream with  
8582 POSIX\_TRACE\_LOOP policy to record trace events continuously, but ideally without losing any  
8583 trace events. The online analyzer process must get the trace events at a mean speed equivalent to  
8584 the recording speed. Should the trace stream become full, a trace stream overrun occurs. This  
8585 condition is detected by getting the status of the active trace stream (function  
8586 *posix\_trace\_get\_status()*) and looking at the member *posix\_stream\_overrun\_status* of the read  
8587 **posix\_stream\_status** structure. In addition, two predefined trace event types are defined:

- 8588 1. The beginning of a trace overflow, to locate the beginning of an overflow when reading a  
8589 trace stream
- 8590 2. The end of a trace overflow, to locate the end of an overflow, when reading a trace stream

8591 As a timestamp is associated with these predefined trace events, it is possible to know the  
8592 duration of the overflow.

#### 8593 **Overrun in Dumping Trace Streams into Trace Logs**

8594 The user lets the trace mechanism dump the trace stream initialized with  
8595 POSIX\_TRACE\_FLUSH policy automatically into a trace log. If the dump operation is slower  
8596 than the recording of trace events, the trace stream can overrun. This condition is detected by  
8597 getting the status of the active trace stream (function *posix\_trace\_get\_status()*) and looking at the  
8598 member *posix\_log\_overrun\_status* of the read **posix\_stream\_status** structure. This overrun  
8599 indicates that the trace mechanism is not able to operate in this mode at this speed. It is the  
8600 responsibility of the user to modify one of the trace parameters (the stream size or the trace  
8601 event type filter, for instance) to avoid such overrun conditions, if overruns are to be prevented.  
8602 The same already predefined trace event types (see **Overrun in Trace Streams Initialized with**  
8603 **POSIX\_TRACE\_LOOP Policy**) are used to detect and to know the duration of an overflow.

#### 8604 **Reading an Active Trace Stream**

8605 Although this trace API allows one to read an active trace stream with log while it is tracing, this  
8606 feature can lead to false overflow origin interpretation: the trace log or the reader of the trace  
8607 stream. Reading from an active trace stream with log is thus non-portable, and has been left  
8608 unspecified.

#### 8609 B.2.12 Data Types

8610 The requirement that additional types defined in this section end in “\_t” was prompted by the  
8611 problem of name space pollution. It is difficult to define a type (where that type is not one  
8612 defined by IEEE Std 1003.1-2001) in one header file and use it in another without adding symbols  
8613 to the name space of the program. To allow implementors to provide their own types, all  
8614 conforming applications are required to avoid symbols ending in “\_t”, which permits the  
8615 implementor to provide additional types. Because a major use of types is in the definition of  
8616 structure members, which can (and in many cases must) be added to the structures defined in  
8617 IEEE Std 1003.1-2001, the need for additional types is compelling.

8618 The types, such as **ushort** and **ulong**, which are in common usage, are not defined in  
 8619 IEEE Std 1003.1-2001 (although **ushort\_t** would be permitted as an extension). They can be  
 8620 added to `<sys/types.h>` using a feature test macro (see Section B.2.2.1 (on page 87)). A suggested  
 8621 symbol for these is `_SYSIII`. Similarly, the types like **u\_short** would probably be best controlled  
 8622 by `_BSD`.

8623 Some of these symbols may appear in other headers; see Section B.2.2.2 (on page 88).

8624 **dev\_t** This type may be made large enough to accommodate host-locality considerations  
 8625 of networked systems.

8626 This type must be arithmetic. Earlier proposals allowed this to be non-arithmetic  
 8627 (such as a structure) and provided a `samefile()` function for comparison.

8628 **gid\_t** Some implementations had separated **gid\_t** from **uid\_t** before POSIX.1 was  
 8629 completed. It would be difficult for them to coalesce them when it was  
 8630 unnecessary. Additionally, it is quite possible that user IDs might be different from  
 8631 group IDs because the user ID might wish to span a heterogeneous network,  
 8632 where the group ID might not.

8633 For current implementations, the cost of having a separate **gid\_t** will be only  
 8634 lexical.

8635 **mode\_t** This type was chosen so that implementations could choose the appropriate  
 8636 integer type, and for compatibility with the ISO C standard. 4.3 BSD uses  
 8637 **unsigned short** and the SVID uses **ushort**, which is the same. Historically, only the  
 8638 low-order sixteen bits are significant.

8639 **nlink\_t** This type was introduced in place of **short** for `st_nlink` (see the `<sys/stat.h>` header)  
 8640 in response to an objection that **short** was too small.

8641 **off\_t** This type is used only in `lseek()`, `fcntl()`, and `<sys/stat.h>`. Many implementations  
 8642 would have difficulties if it were defined as anything other than **long**. Requiring  
 8643 an integer type limits the capabilities of `lseek()` to four gigabytes. The ISO C  
 8644 standard supplies routines that use larger types; see `fgetpos()` and `fsetpos()`. XSI-  
 8645 conformant systems provide the `lseeko()` and `ftello()` functions that use larger  
 8646 types.

8647 **pid\_t** The inclusion of this symbol was controversial because it is tied to the issue of the  
 8648 representation of a process ID as a number. From the point of view of a  
 8649 conforming application, process IDs should be “magic cookies”<sup>1</sup> that are produced  
 8650 by calls such as `fork()`, used by calls such as `waitpid()` or `kill()`, and not otherwise  
 8651 analyzed (except that the sign is used as a flag for certain operations).

8652 The concept of a {PID\_MAX} value interacted with this in early proposals. Treating  
 8653 process IDs as an opaque type both removes the requirement for {PID\_MAX} and  
 8654 allows systems to be more flexible in providing process IDs that span a large range  
 8655 of values, or a small one.

8656 Since the values in **uid\_t**, **gid\_t**, and **pid\_t** will be numbers generally, and  
 8657 potentially both large in magnitude and sparse, applications that are based on

8658 \_\_\_\_\_

8659 1. An historical term meaning: “An opaque object, or token, of determinate size, whose significance is known only to the entity  
 8660 which created it. An entity receiving such a token from the generating entity may only make such use of the ‘cookie’ as is defined  
 8661 and permitted by the supplying entity.”

8662 arrays of objects of this type are unlikely to be fully portable in any case. Solutions  
8663 that treat them as magic cookies will be portable.

8664 {CHILD\_MAX} precludes the possibility of a “toy implementation”, where there  
8665 would only be one process.

8666       **ssize\_t**       This is intended to be a signed analog of **size\_t**. The wording is such that an  
8667 implementation may either choose to use a longer type or simply to use the signed  
8668 version of the type that underlies **size\_t**. All functions that return **ssize\_t** (*read()*  
8669 and *write()*) describe as “implementation-defined” the result of an input exceeding  
8670 {SSIZE\_MAX}. It is recognized that some implementations might have **ints** that  
8671 are smaller than **size\_t**. A conforming application would be constrained not to  
8672 perform I/O in pieces larger than {SSIZE\_MAX}, but a conforming application  
8673 using extensions would be able to use the full range if the implementation  
8674 provided an extended range, while still having a single type-compatible interface.

8675       The symbols **size\_t** and **ssize\_t** are also required in **<unistd.h>** to minimize the  
8676 changes needed for calls to *read()* and *write()*. Implementors are reminded that it  
8677 must be possible to include both **<sys/types.h>** and **<unistd.h>** in the same  
8678 program (in either order) without error.

8679       **uid\_t**       Before the addition of this type, the data types used to represent these values  
8680 varied throughout early proposals. The **<sys/stat.h>** header defined these values as  
8681 type **short**, the **<passwd.h>** file (now **<pwd.h>** and **<grp.h>**) used an **int**, and  
8682 *getuid()* returned an **int**. In response to a strong objection to the inconsistent  
8683 definitions, all the types were switched to **uid\_t**.

8684       In practice, those historical implementations that use varying types of this sort can  
8685 typedef **uid\_t** to **short** with no serious consequences.

8686       The problem associated with this change concerns object compatibility after  
8687 structure size changes. Since most implementations will define **uid\_t** as a short, the  
8688 only substantive change will be a reduction in the size of the **passwd** structure.  
8689 Consequently, implementations with an overriding concern for object  
8690 compatibility can pad the structure back to its current size. For that reason, this  
8691 problem was not considered critical enough to warrant the addition of a separate  
8692 type to POSIX.1.

8693       The types **uid\_t** and **gid\_t** are magic cookies. There is no {UID\_MAX} defined by  
8694 POSIX.1, and no structure imposed on **uid\_t** and **gid\_t** other than that they be  
8695 positive arithmetic types. (In fact, they could be **unsigned char**.) There is no  
8696 maximum or minimum specified for the number of distinct user or group IDs.

## 8697 B.3 System Interfaces

8698 See the RATIONALE sections on the individual reference pages.

### 8699 B.3.1 Examples for Spawn

8700 The following long examples are provided in the Rationale (Informative) volume of  
8701 IEEE Std 1003.1-2001 as a supplement to the reference page for *posix\_spawn()*.

#### 8702 Example Library Implementation of Spawn

8703 The *posix\_spawn()* or *posix\_spawnnp()* functions provide the following:

- 8704 • Simply start a process executing a process image. This is the simplest application for process  
8705 creation, and it may cover most executions of *fork()*.
- 8706 • Support I/O redirection, including pipes.
- 8707 • Run the child under a user and group ID in the domain of the parent.
- 8708 • Run the child at any priority in the domain of the parent.

8709 The *posix\_spawn()* or *posix\_spawnnp()* functions do not cover every possible use of the *fork()*  
8710 function, but they do span the common applications: typical use by a shell and a login utility.

8711 The price for an application is that before it calls *posix\_spawn()* or *posix\_spawnnp()*, the parent  
8712 must adjust to a state that *posix\_spawn()* or *posix\_spawnnp()* can map to the desired state for the  
8713 child. Environment changes require the parent to save some of its state and restore it afterwards.  
8714 The effective behavior of a successful invocation of *posix\_spawn()* is as if the operation were  
8715 implemented with POSIX operations as follows:

```

8716 #include <sys/types.h>
8717 #include <stdlib.h>
8718 #include <stdio.h>
8719 #include <unistd.h>
8720 #include <sched.h>
8721 #include <fcntl.h>
8722 #include <signal.h>
8723 #include <errno.h>
8724 #include <string.h>
8725 #include <signal.h>

8726 /* #include <spawn.h> */
8727 /*****
8728  /* Things that could be defined in spawn.h */
8729  *****/
8730 typedef struct
8731 {
8732     short posix_attr_flags;
8733     #define POSIX_SPAWN_SETPGROUP      0x1
8734     #define POSIX_SPAWN_SETSIGMASK    0x2
8735     #define POSIX_SPAWN_SETSIGDEF     0x4
8736     #define POSIX_SPAWN_SETSCHEDULER  0x8
8737     #define POSIX_SPAWN_SETSCHEDPARAM 0x10
8738     #define POSIX_SPAWN_RESETIDS      0x20
8739     pid_t posix_attr_pgroup;
8740     sigset_t posix_attr_sigmask;
8741     sigset_t posix_attr_sigdefault;
```

```

8742         int posix_attr_schedpolicy;
8743         struct sched_param posix_attr_schedparam;
8744     }    posix_spawnattr_t;

8745     typedef char *posix_spawn_file_actions_t;

8746     int posix_spawn_file_actions_init(
8747         posix_spawn_file_actions_t *file_actions);
8748     int posix_spawn_file_actions_destroy(
8749         posix_spawn_file_actions_t *file_actions);
8750     int posix_spawn_file_actions_addclose(
8751         posix_spawn_file_actions_t *file_actions, int fildes);
8752     int posix_spawn_file_actions_adddup2(
8753         posix_spawn_file_actions_t *file_actions, int fildes,
8754         int newfildes);
8755     int posix_spawn_file_actions_addopen(
8756         posix_spawn_file_actions_t *file_actions, int fildes,
8757         const char *path, int oflag, mode_t mode);
8758     int posix_spawnattr_init(posix_spawnattr_t *attr);
8759     int posix_spawnattr_destroy(posix_spawnattr_t *attr);
8760     int posix_spawnattr_getflags(const posix_spawnattr_t *attr,
8761         short *lags);
8762     int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags);
8763     int posix_spawnattr_getpgroup(const posix_spawnattr_t *attr,
8764         pid_t *pgroup);
8765     int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup);
8766     int posix_spawnattr_getschedpolicy(const posix_spawnattr_t *attr,
8767         int *schedpolicy);
8768     int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr,
8769         int schedpolicy);
8770     int posix_spawnattr_getschedparam(const posix_spawnattr_t *attr,
8771         struct sched_param *schedparam);
8772     int posix_spawnattr_setschedparam(posix_spawnattr_t *attr,
8773         const struct sched_param *schedparam);
8774     int posix_spawnattr_getsigmask(const posix_spawnattr_t *attr,
8775         sigset_t *sigmask);
8776     int posix_spawnattr_setsigmask(posix_spawnattr_t *attr,
8777         const sigset_t *sigmask);
8778     int posix_spawnattr_getdefault(const posix_spawnattr_t *attr,
8779         sigset_t *sigdefault);
8780     int posix_spawnattr_setsigdefault(posix_spawnattr_t *attr,
8781         const sigset_t *sigdefault);
8782     int posix_spawn(pid_t *pid, const char *path,
8783         const posix_spawn_file_actions_t *file_actions,
8784         const posix_spawnattr_t *attrp, char *const argv[],
8785         char *const envp[]);
8786     int posix_spawnnp(pid_t *pid, const char *file,
8787         const posix_spawn_file_actions_t *file_actions,
8788         const posix_spawnattr_t *attrp, char *const argv[],
8789         char *const envp[]);

8790     /*****
8791     /* Example posix_spawn() library routine */
8792     /*****

```

```

8793     int posix_spawn(pid_t *pid,
8794                     const char *path,
8795                     const posix_spawn_file_actions_t *file_actions,
8796                     const posix_spawnattr_t *attrp,
8797                     char *const argv[],
8798                     char *const envp[])
8799     {
8800         /* Create process */
8801         if ((*pid = fork()) == (pid_t) 0)
8802         {
8803             /* This is the child process */
8804             /* Worry about process group */
8805             if (attrp->posix_attr_flags & POSIX_SPAWN_SETPGROUP)
8806             {
8807                 /* Override inherited process group */
8808                 if (setpgid(0, attrp->posix_attr_pgroup) != 0)
8809                 {
8810                     /* Failed */
8811                     exit(127);
8812                 }
8813             }
8814             /* Worry about thread signal mask */
8815             if (attrp->posix_attr_flags & POSIX_SPAWN_SETSIGMASK)
8816             {
8817                 /* Set the signal mask (can't fail) */
8818                 sigprocmask(SIG_SETMASK, &attrp->posix_attr_sigmask, NULL);
8819             }
8820             /* Worry about resetting effective user and group IDs */
8821             if (attrp->posix_attr_flags & POSIX_SPAWN_RESETPIDS)
8822             {
8823                 /* None of these can fail for this case. */
8824                 setuid(getuid());
8825                 setgid(getgid());
8826             }
8827             /* Worry about defaulted signals */
8828             if (attrp->posix_attr_flags & POSIX_SPAWN_SETSIGDEF)
8829             {
8830                 struct sigaction deflt;
8831                 sigset_t all_signals;
8832
8833                 int s;
8834
8835                 /* Construct default signal action */
8836                 deflt.sa_handler = SIG_DFL;
8837                 deflt.sa_flags = 0;
8838
8839                 /* Construct the set of all signals */
8840                 sigfillset(&all_signals);
8841
8842                 /* Loop for all signals */
8843                 for (s = 0; sigismember(&all_signals, s); s++)
8844                 {
8845                     /* Signal to be defaulted? */

```

1

```

8842         if (sigismember(&attrp->posix_attr_sigdefault, s))
8843         {
8844             /* Yes; default this signal */
8845             if (sigaction(s, &deflt, NULL) == -1)
8846             {
8847                 /* Failed */
8848                 exit(127);
8849             }
8850         }
8851     }
8852 }

8853 /* Worry about the fds if they are to be mapped */
8854 if (file_actions != NULL)
8855 {
8856     /* Loop for all actions in object file_actions */
8857     /* (implementation dives beneath abstraction) */
8858     char *p = *file_actions;
8859     while (*p != '\0')
8860     {
8861         if (strncmp(p, "close(", 6) == 0)
8862         {
8863             int fd;
8864             if (sscanf(p + 6, "%d", &fd) != 1)
8865             {
8866                 exit(127);
8867             }
8868             if (close(fd) == -1)
8869                 exit(127);
8870         }
8871         else if (strncmp(p, "dup2(", 5) == 0)
8872         {
8873             int fd, newfd;
8874             if (sscanf(p + 5, "%d,%d", &fd, &newfd) != 2)
8875             {
8876                 exit(127);
8877             }
8878             if (dup2(fd, newfd) == -1)
8879                 exit(127);
8880         }
8881         else if (strncmp(p, "open(", 5) == 0)
8882         {
8883             int fd, oflag;
8884             mode_t mode;
8885             int tempfd;
8886             char path[1000];    /* Should be dynamic */
8887             char *q;
8888             if (sscanf(p + 5, "%d,", &fd) != 1)
8889             {
8890                 exit(127);
8891             }

```

```

8892         p = strchr(p, ',') + 1;
8893         q = strchr(p, '*');
8894         if (q == NULL)
8895             exit(127);
8896         strncpy(path, p, q - p);
8897         path[q - p] = '\0';
8898         if (sscanf(q + 1, "%o,%o", &oflag, &mode) != 2)
8899             {
8900                 exit(127);
8901             }
8902         if (close(fd) == -1)
8903             {
8904                 if (errno != EBADF)
8905                     exit(127);
8906             }
8907         tempfd = open(path, oflag, mode);
8908         if (tempfd == -1)
8909             exit(127);
8910         if (tempfd != fd)
8911             {
8912                 if (dup2(tempfd, fd) == -1)
8913                     {
8914                         exit(127);
8915                     }
8916                 if (close(tempfd) == -1)
8917                     {
8918                         exit(127);
8919                     }
8920             }
8921         }
8922     else
8923     {
8924         exit(127);
8925     }
8926     p = strchr(p, ')') + 1;
8927 }
8928
8929 /* Worry about setting new scheduling policy and parameters */
8930 if (attrp->posix_attr_flags & POSIX_SPAWN_SETSCHEDULER)
8931 {
8932     if (sched_setscheduler(0, attrp->posix_attr_schedpolicy,
8933         &attrp->posix_attr_schedparam) == -1)
8934     {
8935         exit(127);
8936     }
8937 }
8938
8939 /* Worry about setting only new scheduling parameters */
8940 if (attrp->posix_attr_flags & POSIX_SPAWN_SETSCHEDPARAM)
8941 {
8942     if (sched_setparam(0, &attrp->posix_attr_schedparam) == -1)
8943     {

```



```

8943         exit(127);
8944     }
8945 }
8946
8947     /* Now execute the program at path */
8948     /* Any fd that still has FD_CLOEXEC set will be closed */
8949     execve(path, argv, envp);
8950     exit(127);          /* exec failed */
8951 }
8952 else
8953 {
8954     /* This is the parent (calling) process */
8955     if (*pid == (pid_t) - 1)
8956         return errno;
8957     return 0;
8958 }
8959
8960 /* *****
8961 /* Here is a crude but effective implementation of the */
8962 /* file action object operators which store actions as */
8963 /* concatenated token-separated strings.          */
8964 /* *****
8965 /* Create object with no actions. */
8966 int posix_spawn_file_actions_init(
8967     posix_spawn_file_actions_t *file_actions)
8968 {
8969     *file_actions = malloc(sizeof(char));
8970     if (*file_actions == NULL)
8971         return ENOMEM;
8972     strcpy(*file_actions, "");
8973     return 0;
8974 }
8975
8976 /* Free object storage and make invalid. */
8977 int posix_spawn_file_actions_destroy(
8978     posix_spawn_file_actions_t *file_actions)
8979 {
8980     free(*file_actions);
8981     *file_actions = NULL;
8982     return 0;
8983 }
8984
8985 /* Add a new action string to object. */
8986 static int add_to_file_actions(
8987     posix_spawn_file_actions_t *file_actions, char *new_action)
8988 {
8989     *file_actions = realloc
8990     (*file_actions, strlen(*file_actions) + strlen(new_action) + 1);
8991     if (*file_actions == NULL)
8992         return ENOMEM;
8993     strcat(*file_actions, new_action);
8994     return 0;
8995 }

```

```

8993      /* Add a close action to object. */
8994      int posix_spawn_file_actions_addclose(
8995          posix_spawn_file_actions_t *file_actions, int fildes)
8996      {
8997          char temp[100];
8998          sprintf(temp, "close(%d)", fildes);
8999          return add_to_file_actions(file_actions, temp);
9000      }
9001
9002      /* Add a dup2 action to object. */
9003      int posix_spawn_file_actions_adddup2(
9004          posix_spawn_file_actions_t *file_actions, int fildes,
9005          int newfildes)
9006      {
9007          char temp[100];
9008          sprintf(temp, "dup2(%d,%d)", fildes, newfildes);
9009          return add_to_file_actions(file_actions, temp);
9010      }
9011
9012      /* Add an open action to object. */
9013      int posix_spawn_file_actions_addopen(
9014          posix_spawn_file_actions_t *file_actions, int fildes,
9015          const char *path, int oflag, mode_t mode)
9016      {
9017          char temp[100];
9018          sprintf(temp, "open(%d,%s*%o,%o)", fildes, path, oflag, mode);
9019          return add_to_file_actions(file_actions, temp);
9020      }
9021
9022      /* Here is a crude but effective implementation of the */
9023      /* spawn attributes object functions which manipulate */
9024      /* the individual attributes. */
9025      /* ***** */
9026      /* Initialize object with default values. */
9027      int posix_spawnattr_init(posix_spawnattr_t *attr)
9028      {
9029          attr->posix_attr_flags = 0;
9030          attr->posix_attr_pgroup = 0;
9031          /* Default value of signal mask is the parent's signal mask; */
9032          /* other values are also allowed */
9033          sigprocmask(0, NULL, &attr->posix_attr_sigmask);
9034          sigemptyset(&attr->posix_attr_sigdefault);
9035          /* Default values of scheduling attr inherited from the parent; */
9036          /* other values are also allowed */
9037          attr->posix_attr_schedpolicy = sched_getscheduler(0);
9038          sched_getparam(0, &attr->posix_attr_schedparam);
9039          return 0;
9040      }
9041
9042      int posix_spawnattr_destroy(posix_spawnattr_t *attr)
9043      {
9044          /* No action needed */

```

```
9042         return 0;
9043     }
9044     int posix_spawnattr_getflags(const posix_spawnattr_t *attr,
9045         short *flags)
9046     {
9047         *flags = attr->posix_attr_flags;
9048         return 0;
9049     }
9050     int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags)
9051     {
9052         attr->posix_attr_flags = flags;
9053         return 0;
9054     }
9055     int posix_spawnattr_getpgroup(const posix_spawnattr_t *attr,
9056         pid_t *pgroup)
9057     {
9058         *pgroup = attr->posix_attr_pgroup;
9059         return 0;
9060     }
9061     int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup)
9062     {
9063         attr->posix_attr_pgroup = pgroup;
9064         return 0;
9065     }
9066     int posix_spawnattr_getschedpolicy(const posix_spawnattr_t *attr,
9067         int *schedpolicy)
9068     {
9069         *schedpolicy = attr->posix_attr_schedpolicy;
9070         return 0;
9071     }
9072     int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr,
9073         int schedpolicy)
9074     {
9075         attr->posix_attr_schedpolicy = schedpolicy;
9076         return 0;
9077     }
9078     int posix_spawnattr_getschedparam(const posix_spawnattr_t *attr,
9079         struct sched_param *schedparam)
9080     {
9081         *schedparam = attr->posix_attr_schedparam;
9082         return 0;
9083     }
9084     int posix_spawnattr_setschedparam(posix_spawnattr_t *attr,
9085         const struct sched_param *schedparam)
9086     {
9087         attr->posix_attr_schedparam = *schedparam;
9088         return 0;
9089     }
```

```

9090     int posix_spawnattr_getsigmask(const posix_spawnattr_t *attr,
9091                                     sigset_t *sigmask)
9092     {
9093         *sigmask = attr->posix_attr_sigmask;
9094         return 0;
9095     }
9096
9097     int posix_spawnattr_setsigmask(posix_spawnattr_t *attr,
9098                                     const sigset_t *sigmask)
9099     {
9100         attr->posix_attr_sigmask = *sigmask;
9101         return 0;
9102     }
9103
9104     int posix_spawnattr_getsigdefault(const posix_spawnattr_t *attr,
9105                                       sigset_t *sigdefault)
9106     {
9107         *sigdefault = attr->posix_attr_sigdefault;
9108         return 0;
9109     }
9110
9111     int posix_spawnattr_setsigdefault(posix_spawnattr_t *attr,
9112                                       const sigset_t *sigdefault)
9113     {
9114         attr->posix_attr_sigdefault = *sigdefault;
9115         return 0;
9116     }

```

#### 9114 I/O Redirection with Spawn

9115 I/O redirection with *posix\_spawn()* or *posix\_spawnnp()* is accomplished by crafting a *file\_actions*  
 9116 argument to effect the desired redirection. Such a redirection follows the general outline of the  
 9117 following example:

```

9118     /* To redirect new standard output (fd 1) to a file, */
9119     /* and redirect new standard input (fd 0) from my fd socket_pair[1], */
9120     /* and close my fd socket_pair[0] in the new process. */
9121     posix_spawn_file_actions_t file_actions;
9122     posix_spawn_file_actions_init(&file_actions);
9123     posix_spawn_file_actions_addopen(&file_actions, 1, "newout", ...);
9124     posix_spawn_file_actions_dup2(&file_actions, socket_pair[1], 0);
9125     posix_spawn_file_actions_close(&file_actions, socket_pair[0]);
9126     posix_spawn_file_actions_close(&file_actions, socket_pair[1]);
9127     posix_spawn(..., &file_actions, ...);
9128     posix_spawn_file_actions_destroy(&file_actions);

```

**9129        Spawning a Process Under a New User ID**

9130        Spawning a process under a new user ID follows the outline shown in the following example:

```
9131        Save = getuid();  
9132        setuid(newid);  
9133        posix_spawn(...);  
9134        setuid(Save);
```



9136 / *Rationale (Informative)*

9137 **Part C:**

9138 **Shell and Utilities**

9139 *The Open Group*

9140 *The Institute of Electrical and Electronics Engineers, Inc.*





# Rationale for Shell and Utilities

9141

## 9142 C.1 Introduction

### 9143 C.1.1 Scope

9144 Refer to Section A.1.1 (on page 3).

### 9145 C.1.2 Conformance

9146 Refer to Section A.2 (on page 9).

### 9147 C.1.3 Normative References

9148 There is no additional rationale provided for this section.

### 9149 C.1.4 Change History

9150 The change history is provided as an informative section, to track changes from previous issues  
9151 of IEEE Std 1003.1-2001.

9152 The following sections describe changes made to the Shell and Utilities volume of  
9153 IEEE Std 1003.1-2001 since Issue 5 of the base document. The CHANGE HISTORY section for  
9154 each utility describes technical changes made to that utility from Issue 5. Changes between  
9155 earlier issues of the base document and Issue 5 are not included.

9156 The change history between Issue 5 and Issue 6 also lists the changes since the  
9157 ISO POSIX-2: 1993 standard.

#### 9158 Changes from Issue 5 to Issue 6 (IEEE Std 1003.1-2001)

9159 The following list summarizes the major changes that were made in the Shell and Utilities  
9160 volume of IEEE Std 1003.1-2001 from Issue 5 to Issue 6:

9161 • This volume of IEEE Std 1003.1-2001 is extensively revised so that it can be both an IEEE  
9162 POSIX Standard and an Open Group Technical Standard.

9163 • The terminology has been reworked to meet the style requirements.

9164 • Shading notation and margin codes are introduced for identification of options within the  
9165 volume.

9166 • This volume of IEEE Std 1003.1-2001 is updated to mandate support of FIPS 151-2. The  
9167 following changes were made:

9168 — Support is mandated for the capabilities associated with the following symbolic  
9169 constants:

9170 `_POSIX_CHOWN_RESTRICTED`

9171 `_POSIX_JOB_CONTROL`

9172 `_POSIX_SAVED_IDS`

9173 — In the environment for the login shell, the environment variables *LOGNAME* and *HOME*  
9174 shall be defined and have the properties described in the Base Definitions volume of

- 9175 IEEE Std 1003.1-2001, Chapter 7, Locale.
- 9176 • This volume of IEEE Std 1003.1-2001 is updated to align with some features of the Single  
9177 UNIX Specification.
- 9178 • A new section on Utility Limits is added.
- 9179 • A section on the Relationships to Other Documents is added.
- 9180 • Concepts and definitions have been moved to a separate volume.
- 9181 • A RATIONALE section is added to each reference page.
- 9182 • The *c99* utility is added as a replacement for *c89*, which is withdrawn in this issue.
- 9183 • IEEE Std 1003.2d-1994 is incorporated, adding the *qalter*, *qdel*, *qhold*, *qmove*, *qmsg*, *qrerun*, *qrls*,  
9184 *qselect*, *qsig*, *qstat*, and *qsub* utilities.
- 9185 • IEEE P1003.2b draft standard is incorporated, making extensive updates and adding the *iconv*  
9186 utility.
- 9187 • IEEE PASC Interpretations are applied.
- 9188 • The Open Group's corrigenda and resolutions are applied.

### 9189 New Features in Issue 6

9190 The following table lists the new utilities introduced since the ISO POSIX-2: 1993 standard (as  
9191 modified by IEEE Std 1003.2d-1994). Apart from the *c99* and *iconv* utilities, these are all part of  
9192 the XSI extension.

9193  
9194

New Utilities in Issue 6							
<i>admin</i>	<i>compress</i>	<i>gencat</i>	<i>ipcrm</i>	<i>nl</i>	<i>tsort</i>	<i>unlink</i>	<i>val</i>
<i>c99</i>	<i>cxref</i>	<i>get</i>	<i>ipcs</i>	<i>prs</i>	<i>ulimit</i>	<i>uucp</i>	<i>what</i>
<i>cal</i>	<i>delta</i>	<i>hash</i>	<i>link</i>	<i>sact</i>	<i>uncompress</i>	<i>uustat</i>	<i>zcat</i>
<i>cflow</i>	<i>fuser</i>	<i>iconv</i>	<i>m4</i>	<i>sccs</i>	<i>unget</i>	<i>uux</i>	

### 9199 C.1.5 Terminology

9200 Refer to Section A.1.4 (on page 5).

### 9201 C.1.6 Definitions

9202 Refer to Section A.3 (on page 13).

### 9203 C.1.7 Relationship to Other Documents

#### 9204 C.1.7.1 System Interfaces

9205 It has been pointed out that the Shell and Utilities volume of IEEE Std 1003.1-2001 assumes that  
9206 a great deal of functionality from the System Interfaces volume of IEEE Std 1003.1-2001 is  
9207 present, but never states exactly how much (and strictly does not need to since both are  
9208 mandated on a conforming system). This section is an attempt to clarify the assumptions.

9209	<b>File Read, Write, and Creation</b>	2
9210	IEEE Std 1003.1-2001/Cor 2-2004, item XCU/TC2/D6/2 is applied, updating Table 1-1.	2
9211	<b>File Removal</b>	
9212	This is intended to be a summary of the <i>unlink()</i> and <i>rmdir()</i> requirements. Note that it is	
9213	possible using the <i>unlink()</i> function for item 4. to occur.	
9214	<i>C.1.7.2 Concepts Derived from the ISO C Standard</i>	
9215	This section was introduced to address the issue that there was insufficient detail presented by	
9216	such utilities as <i>awk</i> or <i>sh</i> about their procedural control statements and their methods of	
9217	performing arithmetic functions.	
9218	The ISO C standard was selected as a model because most historical implementations of the	
9219	standard utilities were written in C. Thus, it was more likely that they would act in the desired	
9220	manner without modification.	
9221	Using the ISO C standard is primarily a notational convenience so that the many procedural	
9222	languages in the Shell and Utilities volume of IEEE Std 1003.1-2001 would not have to be	
9223	rigorously described in every aspect. Its selection does not require that the standard utilities be	
9224	written in Standard C; they could be written in Common Usage C, Ada, Pascal, assembler	
9225	language, or anything else.	
9226	The sizes of the various numeric values refer to C-language data types that are allowed to be	
9227	different sizes by the ISO C standard. Thus, like a C-language application, a shell application	
9228	cannot rely on their exact size. However, it can rely on their minimum sizes expressed in the	
9229	ISO C standard, such as {LONG_MAX} for a <b>long</b> type.	
9230	The behavior on overflow is undefined for ISO C standard arithmetic. Therefore, the standard	
9231	utilities can use “bignum” representation for integers so that there is no fixed maximum unless	
9232	otherwise stated in the utility description. Similarly, standard utilities can use infinite-precision	
9233	representations for floating-point arithmetic, as long as these representations exceed the ISO C	
9234	standard requirements.	
9235	This section addresses only the issue of semantics; it is not intended to specify syntax. For	
9236	example, the ISO C standard requires that 0L be recognized as an integer constant equal to zero,	
9237	but utilities such as <i>awk</i> and <i>sh</i> are not required to recognize 0L (though they are allowed to, as	
9238	an extension).	
9239	The ISO C standard requires that a C compiler must issue a diagnostic for constants that are too	
9240	large to represent. Most standard utilities are not required to issue these diagnostics; for	
9241	example, the command:	
9242	<code>diff -C 2147483648 file1 file2</code>	
9243	has undefined behavior, and the <i>diff</i> utility is not required to issue a diagnostic even if the	
9244	number 2 147 483 648 cannot be represented.	

9245 **C.1.8 Portability**

9246 Refer to Section A.1.5 (on page 8).

9247 **C.1.8.1 Codes**

9248 Refer to Section A.1.5.1 (on page 8).

9249 **C.1.9 Utility Limits**

9250 This section grew out of an idea that originated with the original POSIX.1, in the tables of system  
 9251 limits for the *sysconf()* and *pathconf()* functions. The idea being that a conforming application  
 9252 can be written to use the most restrictive values that a minimal system can provide, but it should  
 9253 not have to. The values provided represent compromises so that some vendors can use  
 9254 historically limited versions of UNIX system utilities. They are the highest values that a strictly  
 9255 conforming application can assume, given no other information.

9256 However, by using the *getconf* utility or the *sysconf()* function, the elegant application can be  
 9257 tailored to more liberal values on some of the specific instances of specific implementations.

9258 There is no explicitly stated requirement that an implementation provide finite limits for any of  
 9259 these numeric values; the implementation is free to provide essentially unbounded capabilities  
 9260 (where it makes sense), stopping only at reasonable points such as {ULONG\_MAX} (from the  
 9261 ISO C standard). Therefore, applications desiring to tailor themselves to the values on a  
 9262 particular implementation need to be ready for possibly huge values; it may not be a good idea  
 9263 to allocate blindly a buffer for an input line based on the value of {LINE\_MAX}, for instance.  
 9264 However, unlike the System Interfaces volume of IEEE Std 1003.1-2001, there is no set of limits  
 9265 that return a special indication meaning “unbounded”. The implementation should always  
 9266 return an actual number, even if the number is very large.

9267 The statement:

9268 “It is not guaranteed that the application ...”

9269 is an indication that many of these limits are designed to ensure that implementors design their  
 9270 utilities without arbitrary constraints related to unimaginative programming. There are certainly  
 9271 conditions under which combinations of options can cause failures that would not render an  
 9272 implementation non-conforming. For example, {EXPR\_NEST\_MAX} and {ARG\_MAX} could  
 9273 collide when expressions are large; combinations of {BC\_SCALE\_MAX} and {BC\_DIM\_MAX}  
 9274 could exceed virtual memory.

9275 In the Shell and Utilities volume of IEEE Std 1003.1-2001, the notion of a limit being guaranteed  
 9276 for the process lifetime, as it is in the System Interfaces volume of IEEE Std 1003.1-2001, is not as  
 9277 useful to a shell script. The *getconf* utility is probably a process itself, so the guarantee would be  
 9278 without value. Therefore, the Shell and Utilities volume of IEEE Std 1003.1-2001 requires the  
 9279 guarantee to be for the session lifetime. This will mean that many vendors will either return very  
 9280 conservative values or possibly implement *getconf* as a built-in.

9281 It may seem confusing to have limits that apply only to a single utility grouped into one global  
 9282 section. However, the alternative, which would be to disperse them out into their utility  
 9283 description sections, would cause great difficulty when *sysconf()* and *getconf* were described.  
 9284 Therefore, the standard developers chose the global approach.

9285 Each language binding could provide symbol names that are slightly different from those shown  
 9286 here. For example, the C-Language Binding option adds a leading underscore to the symbols as a  
 9287 prefix.

9288 The following comments describe selection criteria for the symbols and their values:

9289 {ARG\_MAX}

9290 This is defined by the System Interfaces volume of IEEE Std 1003.1-2001. Unfortunately, it is

9291 very difficult for a conforming application to deal with this value, as it does not know how

9292 much of its argument space is being consumed by the environment variables of the user.

9293 {BC\_BASE\_MAX}

9294 {BC\_DIM\_MAX}

9295 {BC\_SCALE\_MAX}

9296 These were originally one value, {BC\_SCALE\_MAX}, but it was unreasonable to link all

9297 three concepts into one limit.

9298 {CHILD\_MAX}

9299 This is defined by the System Interfaces volume of IEEE Std 1003.1-2001.

9300 {COLL\_WEIGHTS\_MAX}

9301 The weights assigned to **order** can be considered as “passes” through the collation

9302 algorithm.

9303 {EXPR\_NEST\_MAX}

9304 The value for expression nesting was borrowed from the ISO C standard.

9305 {LINE\_MAX}

9306 This is a global limit that affects all utilities, unless otherwise noted. The {MAX\_CANON}

9307 value from the System Interfaces volume of IEEE Std 1003.1-2001 may further limit input

9308 lines from terminals. The {LINE\_MAX} value was the subject of much debate and is a

9309 compromise between those who wished to have unlimited lines and those who understood

9310 that many historical utilities were written with fixed buffers. Frequently, utility writers

9311 selected the UNIX system constant BUFSIZ to allocate these buffers; therefore, some utilities

9312 were limited to 512 bytes for I/O lines, while others achieved 4 096 bytes or greater.

9313 It should be noted that {LINE\_MAX} applies only to input line length; there is no

9314 requirement in IEEE Std 1003.1-2001 that limits the length of output lines. Utilities such as

9315 *awk*, *sed*, and *paste* could theoretically construct lines longer than any of the input lines they

9316 received, depending on the options used or the instructions from the application. They are

9317 not required to truncate their output to {LINE\_MAX}. It is the responsibility of the

9318 application to deal with this. If the output of one of those utilities is to be piped into another

9319 of the standard utilities, line length restrictions will have to be considered; the *fold* utility,

9320 among others, could be used to ensure that only reasonable line lengths reach utilities or

9321 applications.

9322 {LINK\_MAX}

9323 This is defined by the System Interfaces volume of IEEE Std 1003.1-2001.

9324 {MAX\_CANON}

9325 {MAX\_INPUT}

9326 {NAME\_MAX}

9327 {NGROUPS\_MAX}

9328 {OPEN\_MAX}

9329 {PATH\_MAX}

9330 {PIPE\_BUF}

9331 These limits are defined by the System Interfaces volume of IEEE Std 1003.1-2001. Note that

9332 the byte lengths described by some of these values continue to represent bytes, even if the

9333 applicable character set uses a multi-byte encoding.

9334 {RE\_DUP\_MAX}  
 9335 The value selected is consistent with historical practice. Although the name implies that it  
 9336 applies to all REs, only BREs use the interval notation `\{m,n\}` addressed by this limit.

9337 {POSIX2\_SYMLINKS}  
 9338 The {POSIX2\_SYMLINKS} variable indicates that the underlying operating system supports  
 9339 the creation of symbolic links in specific directories. Many of the utilities defined in  
 9340 IEEE Std 1003.1-2001 that deal with symbolic links do not depend on this value. For  
 9341 example, a utility that follows symbolic links (or does not, as the case may be) will only be  
 9342 affected by a symbolic link if it encounters one. Presumably, a file system that does not  
 9343 support symbolic links will not contain any. This variable does affect such utilities as *ln -s*  
 9344 and *pax* that attempt to create symbolic links. 2

9345 There are different limits associated with command lines and input to utilities, depending on the  
 9346 method of invocation. In the case of a C program *exec-ing* a utility, {ARG\_MAX} is the  
 9347 underlying limit. In the case of the shell reading a script and *exec-ing* a utility, {LINE\_MAX}  
 9348 limits the length of lines the shell is required to process, and {ARG\_MAX} will still be a limit. If a  
 9349 user is entering a command on a terminal to the shell, requesting that it invoke the utility,  
 9350 {MAX\_INPUT} may restrict the length of the line that can be given to the shell to a value below  
 9351 {LINE\_MAX}.

9352 When an option is supported, *getconf* returns a value of 1. For example, when C development is  
 9353 supported:

```
9354     if [ "$(getconf POSIX2_C_DEV)" -eq 1 ]; then
9355         echo C supported
9356     fi
```

9357 The *sysconf()* function in the C-Language Binding option would return 1.

9358 The following comments describe selection criteria for the symbols and their values:

9359 POSIX2\_C\_BIND  
 9360 POSIX2\_C\_DEV  
 9361 POSIX2\_FORT\_DEV  
 9362 POSIX2\_FORT\_RUN  
 9363 POSIX2\_SW\_DEV  
 9364 POSIX2\_UPE

9365 It is possible for some (usually privileged) operations to remove utilities that support these  
 9366 options or otherwise to render these options unsupported. The header files, the *sysconf()*  
 9367 function, or the *getconf* utility will not necessarily detect such actions, in which case they  
 9368 should not be considered as rendering the implementation non-conforming. A test suite  
 9369 should not attempt tests such as:

```
9370     rm /usr/bin/c99
9371     getconf POSIX2_C_DEV
```

9372 POSIX2\_LOCALEDEF

9373 This symbol was introduced to allow implementations to restrict supported locales to only  
 9374 those supplied by the implementation.

9375 IEEE Std 1003.1-2001/Cor 1-2002, item XCU/TC1/D6/2 is applied, deleting the entry for 1  
 9376 {POSIX2\_VERSION} since it is not a utility limit minimum value. 1

9377 IEEE Std 1003.1-2001/Cor 1-2002, item XCU/TC1/D6/3 is applied, changing the text in Utility 1  
 9378 Limits from: “utility (see *getconf*) through the *sysconf()* function defined in the System Interfaces 1  
 9379 volume of IEEE Std 1003.1-2001. The literal names shown in Table 1-3 apply only to the *getconf* 1  
 9380 utility; the high-level language binding describes the exact form of each name to be used by the 1

9381 interfaces in that binding.” to: “utility (see *getconf*).”.

#### 9382 **C.1.10 Grammar Conventions**

9383 There is no additional rationale provided for this section.

#### 9384 **C.1.11 Utility Description Defaults**

9385 This section is arranged with headings in the same order as all the utility descriptions. It is a  
9386 collection of related and unrelated information concerning:

- 9387 1. The default actions of utilities
- 9388 2. The meanings of notations used in IEEE Std 1003.1-2001 that are specific to individual  
9389 utility sections

9390 Although this material may seem out of place here, it is important that this information appear  
9391 before any of the utilities to be described later.

#### 9392 **NAME**

9393 There is no additional rationale provided for this section.

#### 9394 **SYNOPSIS**

9395 There is no additional rationale provided for this section.

#### 9396 **DESCRIPTION**

9397 There is no additional rationale provided for this section.

#### 9398 **OPTIONS**

9399 Although it has not always been possible, the standard developers tried to avoid repeating  
9400 information to reduce the risk that duplicate explanations could each be modified differently.

9401 The need to recognize `--` is required because conforming applications need to shield their  
9402 operands from any arbitrary options that the implementation may provide as an extension. For  
9403 example, if the standard utility *foo* is listed as taking no options, and the application needed to  
9404 give it a pathname with a leading hyphen, it could safely do it as:

9405 `foo -- -myfile`

9406 and avoid any problems with `-m` used as an extension.

#### 9407 **OPERANDS**

9408 The usage of `-` is never shown in the SYNOPSIS. Similarly, the usage of `--` is never shown.

9409 The requirement for processing operands in command-line order is to avoid a “WeirdNIX”  
9410 utility that might choose to sort the input files alphabetically, by size, or by directory order.  
9411 Although this might be acceptable for some utilities, in general the programmer has a right to  
9412 know exactly what order will be chosen.

9413 Some of the standard utilities take multiple *file* operands and act as if they were processing the  
9414 concatenation of those files. For example:

9415 `asa file1 file2`

9416 and:

9417 `cat file1 file2 | asa`

9418 have similar results when questions of file access, errors, and performance are ignored. Other  
 9419 utilities such as *grep* or *wc* have completely different results in these two cases. This latter type of  
 9420 utility is always identified in its DESCRIPTION or OPERANDS sections, whereas the former is  
 9421 not. Although it might be possible to create a general assertion about the former case, the  
 9422 following points must be addressed:

- 9423 • Access times for the files might be different in the operand case *versus* the *cat* case.
- 9424 • The utility may have error messages that are cognizant of the input filename, and this added  
 9425 value should not be suppressed. (As an example, *awk* sets a variable with the filename at  
 9426 each file boundary.)

## 9427 STDIN

9428 There is no additional rationale provided for this section.

## 9429 INPUT FILES

9430 A conforming application cannot assume the following three commands are equivalent:

```
9431 tail -n +2 file
9432 (sed -n 1q; cat) < file
9433 cat file | (sed -n 1q; cat)
```

9434 The second command is equivalent to the first only when the file is seekable. In the third  
 9435 command, if the file offset in the open file description were not unspecified, *sed* would have to be  
 9436 implemented so that it read from the pipe 1 byte at a time or it would have to employ some  
 9437 method to seek backwards on the pipe. Such functionality is not defined currently in POSIX.1  
 9438 and does not exist on all historical systems. Other utilities, such as *head*, *read*, and *sh*, have similar  
 9439 properties, so the restriction is described globally in this section.

9440 The definition of “text file” is strictly enforced for input to the standard utilities; very few of  
 9441 them list exceptions to the undefined results called for here. (Of course, “undefined” here does  
 9442 not mean that historical implementations necessarily have to change to start indicating error  
 9443 conditions. Conforming applications cannot rely on implementations succeeding or failing when  
 9444 non-text files are used.)

9445 The utilities that allow line continuation are generally those that accept input languages, rather  
 9446 than pure data. It would be unusual for an input line of this type to exceed {LINE\_MAX} bytes  
 9447 and unreasonable to require that the implementation allow unlimited accumulation of multiple  
 9448 lines, each of which could reach {LINE\_MAX}. Thus, for a conforming application the total of all  
 9449 the continued lines in a set cannot exceed {LINE\_MAX}.

9450 The format description is intended to be sufficiently rigorous to allow other applications to  
 9451 generate these input files. However, since <blank>s can legitimately be included in some of the  
 9452 fields described by the standard utilities, particularly in locales other than the POSIX locale, this  
 9453 intent is not always realized.



**ENVIRONMENT VARIABLES**

There is no additional rationale provided for this section.

**ASYNCHRONOUS EVENTS**

Because there is no language prohibiting it, a utility is permitted to catch a signal, perform some additional processing (such as deleting temporary files), restore the default signal action (or action inherited from the parent process), and resignal itself.

**STDOUT**

The format description is intended to be sufficiently rigorous to allow post-processing of output by other programs, particularly by an *awk* or *lex* parser.

**STDERR**

This section does not describe error messages that refer to incorrect operation of the utility. Consider a utility that processes program source code as its input. This section is used to describe messages produced by a correctly operating utility that encounters an error in the program source code on which it is processing. However, a message indicating that the utility had insufficient memory in which to operate would not be described.

Some utilities have traditionally produced warning messages without returning a non-zero exit status; these are specifically noted in their sections. Other utilities shall not write to standard error if they complete successfully, unless the implementation provides some sort of extension to increase the verbosity or debugging level.

The format descriptions are intended to be sufficiently rigorous to allow post-processing of output by other programs.

**OUTPUT FILES**

The format description is intended to be sufficiently rigorous to allow post-processing of output by other programs, particularly by an *awk* or *lex* parser.

Receipt of the SIGQUIT signal should generally cause termination (unless in some debugging mode) that would bypass any attempted recovery actions.

**EXTENDED DESCRIPTION**

There is no additional rationale provided for this section.

**EXIT STATUS**

Note the additional discussion of exit values in *Exit Status for Commands* in the *sh* utility. It describes requirements for returning exit values greater than 125.

A utility may list zero as a successful return, 1 as a failure for a specific reason, and greater than 1 as “an error occurred”. In this case, unspecified conditions may cause a 2 or 3, or other value, to be returned. A strictly conforming application should be written so that it tests for successful exit status values (zero in this case), rather than relying upon the single specific error value listed in IEEE Std 1003.1-2001. In that way, it will have maximum portability, even on implementations with extensions.

The standard developers are aware that the general non-enumeration of errors makes it difficult to write test suites that test the *incorrect* operation of utilities. There are some historical implementations that have expended effort to provide detailed status messages and a helpful

9494 environment to bypass or explain errors, such as prompting, retrying, or ignoring unimportant  
9495 syntax errors; other implementations have not. Since there is no realistic way to mandate system  
9496 behavior in cases of undefined application actions or system problems—in a manner acceptable  
9497 to all cultures and environments—attention has been limited to the correct operation of utilities  
9498 by the conforming application. Furthermore, the conforming application does not need detailed  
9499 information concerning errors that it caused through incorrect usage or that it cannot correct.

9500 There is no description of defaults for this section because all of the standard utilities specify  
9501 something (or explicitly state “Unspecified”) for exit status.

## 9502 CONSEQUENCES OF ERRORS

9503 Several actions are possible when a utility encounters an error condition, depending on the  
9504 severity of the error and the state of the utility. Included in the possible actions of various  
9505 utilities are: deletion of temporary or intermediate work files; deletion of incomplete files; and  
9506 validity checking of the file system or directory.

9507 The text about recursive traversing is meant to ensure that utilities such as *find* process as many  
9508 files in the hierarchy as they can. They should not abandon all of the hierarchy at the first error  
9509 and resume with the next command-line operand, but should attempt to keep going.

## 9510 APPLICATION USAGE

9511 This section provides additional caveats, issues, and recommendations to the developer.

## 9512 EXAMPLES

9513 This section provides sample usage.

## 9514 RATIONALE

9515 There is no additional rationale provided for this section.

## 9516 FUTURE DIRECTIONS

9517 FUTURE DIRECTIONS sections act as pointers to related work that may impact the interface in  
9518 the future, and often cautions the developer to architect the code to account for a change in this  
9519 area. Note that a future directions statement should not be taken as a commitment to adopt a  
9520 feature or interface in the future.

## 9521 SEE ALSO

9522 There is no additional rationale provided for this section.

## 9523 CHANGE HISTORY

9524 There is no additional rationale provided for this section.

## 9525 C.1.12 Considerations for Utilities in Support of Files of Arbitrary Size

9526 This section is intended to clarify the requirements for utilities in support of large files.

9527 The utilities listed in this section are utilities which are used to perform administrative tasks  
 9528 such as to create, move, copy, remove, change the permissions, or measure the resources of a  
 9529 file. They are useful both as end-user tools and as utilities invoked by applications during  
 9530 software installation and operation.

9531 The *chgrp*, *chmod*, *chown*, *ln*, and *rm* utilities probably require use of large file-capable versions of  
 9532 *stat()*, *lstat()*, *ftw()*, and the **stat** structure.

9533 The *cat*, *cksum*, *cmp*, *cp*, *dd*, *mv*, *sum*, and *touch* utilities probably require use of large file-capable  
 9534 versions of *creat()*, *open()*, and *fopen()*.

9535 The *cat*, *cksum*, *cmp*, *dd*, *df*, *du*, *ls*, and *sum* utilities may require writing large integer values. For  
 9536 example:

- 9537 • The *cat* utility might have a **-n** option which counts <newline>s.
- 9538 • The *cksum* and *ls* utilities report file sizes.
- 9539 • The *cmp* utility reports the line number at which the first difference occurs, and also has a **-l**  
 9540 option which reports file offsets.
- 9541 • The *dd*, *df*, *du*, *ls*, and *sum* utilities report block counts.

9542 The *dd*, *find*, and *test* utilities may need to interpret command arguments that contain 64-bit  
 9543 values. For *dd*, the arguments include *skip=n*, *seek=n*, and *count=n*. For *find*, the arguments  
 9544 include **-sizen**. For *test*, the arguments are those associated with algebraic comparisons.

9545 The *df* utility might need to access large file systems with *statvfs()*.

9546 The *ulimit* utility will need to use large file-capable versions of *getrlimit()* and *setrlimit()* and be  
 9547 able to read and write large integer values.

## 9548 C.1.13 Built-In Utilities

9549 All of these utilities can be *exec*-ed. There is no requirement that these utilities are actually built  
 9550 into the shell itself, but many shells need the capability to do so because the Shell and Utilities  
 9551 volume of IEEE Std 1003.1-2001, Section 2.9.1.1, Command Search and Execution requires that  
 9552 they be found prior to the *PATH* search. The shell could satisfy its requirements by keeping a list  
 9553 of the names and directly accessing the file-system versions regardless of *PATH*. Providing all of  
 9554 the required functionality for those such as *cd* or *read* would be more difficult.

9555 There were originally three justifications for allowing the omission of *exec*-able versions:

- 9556 1. It would require wasting space in the file system, at the expense of very small systems.  
 9557 However, it has been pointed out that all 16 utilities in the table can be provided with 16  
 9558 links to a single-line shell script:

9559 \$0 "\$@"

- 9560 2. It is not logical to require invocation of utilities such as *cd* because they have no value  
 9561 outside the shell environment or cannot be useful in a child process. However, counter-  
 9562 examples always seemed to be available for even the most unusual cases:

9563 find . -type d -exec cd {} \; -exec foo {} \;  
 9564 (which invokes “foo” on accessible directories)

9565 ps ... | sed ... | xargs kill

`find . -exec true \; -a ...`

(where “true” is used for temporary debugging)

3. It is confusing to have a utility such as *kill* that can easily be in the file system in the base standard, but that requires built-in status for the User Portability Utilities option (for the % job control job ID notation). It was decided that it was more appropriate to describe the required functionality (rather than the implementation) to the system implementors and let them decide how to satisfy it.

On the other hand, it was realized that any distinction like this between utilities was not useful to applications, and that the cost to correct it was small. These arguments were ultimately the most effective.

There were varying reasons for including utilities in the table of built-ins:

*alias, fc, unalias*

The functionality of these utilities is performed more simply within the shell itself and that is the model most historical implementations have used.

*bg, fg, jobs*

All of the job control-related utilities are eligible for built-in status because that is the model most historical implementations have used.

*cd, getopts, newgrp, read, umask, wait*

The functionality of these utilities is performed more simply within the context of the current process. An example can be taken from the usage of the *cd* utility. The purpose of the *cd* utility is to change the working directory for subsequent operations. The actions of *cd* affect the process in which *cd* is executed and all subsequent child processes of that process. Based on the POSIX standard process model, changes in the process environment of a child process have no effect on the parent process. If the *cd* utility were executed from a child process, the working directory change would be effective only in the child process. Child processes initiated subsequent to the child process that executed the *cd* utility would not have a changed working directory relative to the parent process.

*command*

This utility was placed in the table primarily to protect scripts that are concerned about their *PATH* being manipulated. The “secure” shell script example in the *command* utility in the Shell and Utilities volume of IEEE Std 1003.1-2001 would not be possible if a *PATH* change retrieved an alien version of *command*. (An alternative would have been to implement *getconf* as a built-in, but the standard developers considered that it carried too many changing configuration strings to require in the shell.)

*kill* Since *kill* provides optional job control functionality using shell notation (%1, %2, and so on), some implementations would find it extremely difficult to provide this outside the shell.

*true, false*

These are in the table as a courtesy to programmers who wish to use the “while true” shell construct without protecting *true* from *PATH* searches. (It is acknowledged that “while :” also works, but the idiom with *true* is historically pervasive.)

All utilities, including those in the table, are accessible via the *system()* and *popen()* functions in the System Interfaces volume of IEEE Std 1003.1-2001. There are situations where the return functionality of *system()* and *popen()* is not desirable. Applications that require the exit status of the invoked utility will not be able to use *system()* or *popen()*, since the exit status returned is that of the command language interpreter rather than that of the invoked utility. The alternative for such applications is the use of the *exec* family.

## 9612 C.2 Shell Command Language

### 9613 C.2.1 Shell Introduction

9614 The System V shell was selected as the starting point for the Shell and Utilities volume of  
 9615 IEEE Std 1003.1-2001. The BSD C shell was excluded from consideration for the following  
 9616 reasons:

- 9617 • Most historically portable shell scripts assume the Version 7 Bourne shell, from which the  
 9618 System V shell is derived.
- 9619 • The majority of tutorial materials on shell programming assume the System V shell.

9620 The construct "#!" is reserved for implementations wishing to provide that extension. If it were  
 9621 not reserved, the Shell and Utilities volume of IEEE Std 1003.1-2001 would disallow it by forcing  
 9622 it to be a comment. As it stands, a strictly conforming application must not use "#!" as the first  
 9623 two characters of the file.

### 9624 C.2.2 Quoting

9625 There is no additional rationale provided for this section.

#### 9626 C.2.2.1 Escape Character (Backslash)

9627 There is no additional rationale provided for this section.

#### 9628 C.2.2.2 Single-Quotes

9629 A backslash cannot be used to escape a single-quote in a single-quoted string. An embedded  
 9630 quote can be created by writing, for example: "'a'\''b'", which yields "a'b". (See the Shell  
 9631 and Utilities volume of IEEE Std 1003.1-2001, Section 2.6.5, Field Splitting for a better  
 9632 understanding of how portions of words are either split into fields or remain concatenated.) A  
 9633 single token can be made up of concatenated partial strings containing all three kinds of quoting  
 9634 or escaping, thus permitting any combination of characters.

#### 9635 C.2.2.3 Double-Quotes

9636 The escaped <newline> used for line continuation is removed entirely from the input and is not  
 9637 replaced by any white space. Therefore, it cannot serve as a token separator.

9638 In double-quoting, if a backslash is immediately followed by a character that would be  
 9639 interpreted as having a special meaning, the backslash is deleted and the subsequent character is  
 9640 taken literally. If a backslash does not precede a character that would have a special meaning, it  
 9641 is left in place unmodified and the character immediately following it is also left unmodified.  
 9642 Thus, for example:

9643       "\\$"    →   \$

9644       "a"     →   \a

9645 It would be desirable to include the statement “The characters from an enclosed "\${" to the  
 9646 matching '}' shall not be affected by the double quotes”, similar to the one for "\$()".  
 9647 However, historical practice in the System V shell prevents this.

9648 The requirement that double-quotes be matched inside "\${...}" within double-quotes and the  
 9649 rule for finding the matching '}' in the Shell and Utilities volume of IEEE Std 1003.1-2001,  
 9650 Section 2.6.2, Parameter Expansion eliminate several subtle inconsistencies in expansion for  
 9651 historical shells in rare cases; for example:

9652           `"${foo-bar}"`

9653           yields **bar** when **foo** is not defined, and is an invalid substitution when **foo** is defined, in many  
 9654           historical shells. The differences in processing the `"${...}"` form have led to inconsistencies  
 9655           between historical systems. A consequence of this rule is that single-quotes cannot be used to  
 9656           quote the `'` within `"${...}"`; for example:

9657           `unset bar`  
 9658           `foo="${bar-'}'"`

9659           is invalid because the `"${...}"` substitution contains an unpaired unescaped single-quote. The  
 9660           backslash can be used to escape the `'` in this example to achieve the desired result:

9661           `unset bar`  
 9662           `foo="${bar-\\}'"`

9663           The differences in processing the `"${...}"` form have led to inconsistencies between the  
 9664           historical System V shell, BSD, and KornShells, and the text in the Shell and Utilities volume of  
 9665           IEEE Std 1003.1-2001 is an attempt to converge them without breaking too many applications.  
 9666           The only alternative to this compromise between shells would be to make the behavior  
 9667           unspecified whenever the literal characters `'`, `'{`, `'`, and `'"` appear within `"${...}"`.  
 9668           To write a portable script that uses these values, a user would have to assign variables; for  
 9669           example:

9670           `squote=\' dquote=\" lbrace='{ rbrace='}'`  
 9671           `${foo-`${squote}${rbrace}${squote}}`

9672           rather than:

9673           `${foo-"' }'"`

9674           Some implementations have allowed the end of the word to terminate the backquoted command  
 9675           substitution, such as in:

9676           `"`echo hello"`

9677           This usage is undefined; the matching backquote is required by the Shell and Utilities volume of  
 9678           IEEE Std 1003.1-2001. The other undefined usage can be illustrated by the example:

9679           `sh -c '` echo "foo`'`

9680           The description of the recursive actions involving command substitution can be illustrated with  
 9681           an example. Upon recognizing the introduction of command substitution, the shell parses input  
 9682           (in a new context), gathering the source for the command substitution until an unbalanced `' )`  
 9683           or `'`` is located. For example, in the following:

9684           `echo "$(date; echo "`  
 9685           `one" )"`

9686           the double-quote following the *echo* does not terminate the first double-quote; it is part of the  
 9687           command substitution script. Similarly, in:

9688           `echo "$(echo *)"`

9689           the asterisk is not quoted since it is inside command substitution; however:

9690           `echo "$(echo "*)"`

9691           is quoted (and represents the asterisk character itself).

9692 **C.2.3 Token Recognition**

9693 The "(" and ")" symbols are control operators in the KornShell, used for an alternative  
 9694 syntax of an arithmetic expression command. A conforming application cannot use "(" as a  
 9695 single token (with the exception of the "\$(" form for shell arithmetic).

9696 On some implementations, the symbol "(" is a control operator; its use produces unspecified  
 9697 results. Applications that wish to have nested subshells, such as:

9698 `((echo Hello);(echo World))`

9699 must separate the "(" characters into two tokens by including white space between them.  
 9700 Some systems may treat these as invalid arithmetic expressions instead of subshells.

9701 Certain combinations of characters are invalid in portable scripts, as shown in the grammar.  
 9702 Implementations may use these combinations (such as "|&") as valid control operators. Portable  
 9703 scripts cannot rely on receiving errors in all cases where this volume of IEEE Std 1003.1-2001  
 9704 indicates that a syntax is invalid.

9705 The (3) rule about combining characters to form operators is not meant to preclude systems from  
 9706 extending the shell language when characters are combined in otherwise invalid ways.  
 9707 Conforming applications cannot use invalid combinations, and test suites should not penalize  
 9708 systems that take advantage of this fact. For example, the unquoted combination "|&" is not  
 9709 valid in a POSIX script, but has a specific KornShell meaning.

9710 The (10) rule about '#' as the current character is the first in the sequence in which a new token  
 9711 is being assembled. The '#' starts a comment only when it is at the beginning of a token. This  
 9712 rule is also written to indicate that the search for the end-of-comment does not consider escaped  
 9713 <newline> specially, so that a comment cannot be continued to the next line.

9714 **C.2.3.1 Alias Substitution**

9715 The alias capability was added in the User Portability Utilities option because it is widely used in  
 9716 historical implementations by interactive users.

9717 The definition of "alias name" precludes an alias name containing a slash character. Since the  
 9718 text applies to the command words of simple commands, reserved words (in their proper  
 9719 places) cannot be confused with aliases.

9720 The placement of alias substitution in token recognition makes it clear that it precedes all of the  
 9721 word expansion steps.

9722 An example concerning trailing <blank>s and reserved words follows. If the user types:

9723 `$ alias foo="/bin/ls "`  
 9724 `$ alias while="/"`

9725 The effect of executing:

9726 `$ while true`  
 9727 `> do`  
 9728 `> echo "Hello, World"`  
 9729 `> done`

9730 is a never-ending sequence of "Hello, World" strings to the screen. However, if the user  
 9731 types:

9732 `$ foo while`

9733 the result is an *ls* listing of /. Since the alias substitution for **foo** ends in a <space>, the next word  
 9734 is checked for alias substitution. The next word, **while**, has also been aliased, so it is substituted

9735 as well. Since it is not in the proper position as a command word, it is not recognized as a  
 9736 reserved word.

9737 If the user types:

9738 `$ foo; while`

9739 **while** retains its normal reserved-word properties.

#### 9740 C.2.4 Reserved Words

9741 All reserved words are recognized syntactically as such in the contexts described. However, note  
 9742 that **in** is the only meaningful reserved word after a **case** or **for**; similarly, **in** is not meaningful as  
 9743 the first word of a simple command.

9744 Reserved words are recognized only when they are delimited (that is, meet the definition of the  
 9745 Base Definitions volume of IEEE Std 1003.1-2001, Section 3.435, Word), whereas operators are  
 9746 themselves delimiters. For instance, `' ( ' and ' ) '` are control operators, so that no `<space>` is  
 9747 needed in (*list*). However, `' { ' and ' } '` are reserved words in { *list*; }, so that in this case the  
 9748 leading `<space>` and semicolon are required.

9749 The list of unspecified reserved words is from the KornShell, so conforming applications cannot  
 9750 use them in places a reserved word would be recognized. This list contained **time** in early  
 9751 proposals, but it was removed when the *time* utility was selected for the Shell and Utilities  
 9752 volume of IEEE Std 1003.1-2001.

9753 There was a strong argument for promoting braces to operators (instead of reserved words), so  
 9754 they would be syntactically equivalent to subshell operators. Concerns about compatibility  
 9755 outweighed the advantages of this approach. Nevertheless, conforming applications should  
 9756 consider quoting `' { ' and ' } '` when they represent themselves.

9757 The restriction on ending a name with a colon is to allow future implementations that support  
 9758 named labels for flow control; see the RATIONALE for the *break* built-in utility.

9759 It is possible that a future version of the Shell and Utilities volume of IEEE Std 1003.1-2001 may  
 9760 require that `' { ' and ' } '` be treated individually as control operators, although the token `" { } "`  
 9761 will probably be a special-case exemption from this because of the often-used *find*{ } construct.

#### 9762 C.2.5 Parameters and Variables

##### 9763 C.2.5.1 Positional Parameters

9764 There is no additional rationale provided for this section.

##### 9765 C.2.5.2 Special Parameters

9766 Most historical implementations implement subshells by forking; thus, the special parameter  
 9767 `' $ '` does not necessarily represent the process ID of the shell process executing the commands  
 9768 since the subshell execution environment preserves the value of `' $ '`.

9769 If a subshell were to execute a background command, the value of `" $ ! "` for the parent would  
 9770 not change. For example:

```
9771 (
9772   date &
9773   echo $!
9774 )
9775 echo $!
```



9776 would echo two different values for "\$!".

9777 The "\$-" special parameter can be used to save and restore *set* options:

```
9778     Save=$(echo $- | sed 's/[ics]//g')
9779     ...
9780     set +aCefnuvx
9781     if [ -n "$Save" ]; then
9782         set -$Save
9783     fi
```

9784 The three options are removed using *sed* in the example because they may appear in the value of  
9785 "\$-" (from the *sh* command line), but are not valid options to *set*.

9786 The descriptions of parameters '\*' and '@' assume the reader is familiar with the field splitting  
9787 discussion in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.6.5, Field Splitting  
9788 and understands that portions of the word remain concatenated unless there is some reason to  
9789 split them into separate fields.

9790 Some examples of the '\*' and '@' properties, including the concatenation aspects:

```
9791     set "abc" "def ghi" "jkl"
9792     echo $*      => "abc" "def" "ghi" "jkl"
9793     echo "$*"    => "abc def ghi jkl"
9794     echo $@      => "abc" "def" "ghi" "jkl"
```

9795 but:

```
9796     echo "$@"    => "abc" "def ghi" "jkl"
9797     echo "xx$@yy" => "xxabc" "def ghi" "jkl"
9798     echo "$@$@"  => "abc" "def ghi" "jklabc" "def ghi" "jkl"
```

9799 In the preceding examples, the double-quote characters that appear after the "=>" do not appear  
9800 in the output and are used only to illustrate word boundaries.

9801 The following example illustrates the effect of setting *IFS* to a null string:

```
9802     $ IFS=''
9803     $ set foo bar bam
9804     $ echo "$@"
9805     foo bar bam
9806     $ echo "$*"
9807     foobarbam
9808     $ unset IFS
9809     $ echo "$*"
9810     foo bar bam
```

### 9811 C.2.5.3 Shell Variables

9812 See the discussion of *IFS* in Section C.2.6.5 (on page 244) and the RATIONALE for the *sh* utility.

9813 The prohibition on *LC\_CTYPE* changes affecting lexical processing protects the shell  
9814 implementor (and the shell programmer) from the ill effects of changing the definition of  
9815 <blank> or the set of alphabetic characters in the current environment. It would probably not be  
9816 feasible to write a compiled version of a shell script without this rule. The rule applies only to  
9817 the current invocation of the shell and its subshells—invoking a shell script or performing *exec sh*  
9818 would subject the new shell to the changes in *LC\_CTYPE*.

9819		Other common environment variables used by historical shells are not specified by the Shell and
9820		Utilities volume of IEEE Std 1003.1-2001, but they should be reserved for the historical uses.
9821		Tilde expansion for components of <i>PATH</i> in an assignment such as:
9822		<code>PATH=~hlj/bin:~dwc/bin:\$PATH</code>
9823		is a feature of some historical shells and is allowed by the wording of the Shell and Utilities
9824		volume of IEEE Std 1003.1-2001, Section 2.6.1, Tilde Expansion. Note that the tildes are expanded
9825		during the assignment to <i>PATH</i> , not when <i>PATH</i> is accessed during command search.
9826		The following entries represent additional information about variables included in the Shell and
9827		Utilities volume of IEEE Std 1003.1-2001, or rationale for common variables in use by shells that
9828		have been excluded:
9829	—	(Underscore.) While underscore is historical practice, its overloaded usage in
9830		the KornShell is confusing, and it has been omitted from the Shell and Utilities
9831		volume of IEEE Std 1003.1-2001.
9832	<i>ENV</i>	This variable can be used to set aliases and other items local to the invocation
9833		of a shell. The file referred to by <i>ENV</i> differs from <i>\$HOME/.profile</i> in that
9834		<i>.profile</i> is typically executed at session start-up, whereas the <i>ENV</i> file is
9835		executed at the beginning of each shell invocation. The <i>ENV</i> value is
9836		interpreted in a manner similar to a dot script, in that the commands are
9837		executed in the current environment and the file needs to be readable, but not
9838		executable. However, unlike dot scripts, no <i>PATH</i> searching is performed.
9839		This is used as a guard against Trojan Horse security breaches.
9840	<i>ERRNO</i>	This variable was omitted from the Shell and Utilities volume of
9841		IEEE Std 1003.1-2001 because the values of error numbers are not defined in
9842		IEEE Std 1003.1-2001 in a portable manner.
9843	<i>FCEDIT</i>	Since this variable affects only the <i>fc</i> utility, it has been omitted from this more
9844		global place. The value of <i>FCEDIT</i> does not affect the command-line editing
9845		mode in the shell; see the description of <i>set -o vi</i> in the <i>set</i> built-in utility.
9846	<i>PS1</i>	This variable is used for interactive prompts. Historically, the “superuser”
9847		has had a prompt of ‘#’. Since privileges are not required to be monolithic, it
9848		is difficult to define which privileges should cause the alternate prompt.
9849		However, a sufficiently powerful user should be reminded of that power by
9850		having an alternate prompt.
9851	<i>PS3</i>	This variable is used by the KornShell for the <i>select</i> command. Since the POSIX
9852		shell does not include <i>select</i> , <i>PS3</i> was omitted.
9853	<i>PS4</i>	This variable is used for shell debugging. For example, the following script:
9854		<code>PS4='[ \${LINENO} ]+ '</code>
9855		<code>set -x</code>
9856		<code>echo Hello</code>
9857		writes the following to standard error:
9858		<code>[3]+ echo Hello</code>
9859	<i>RANDOM</i>	This pseudo-random number generator was not seen as being useful to
9860		interactive users.
9861	<i>SECONDS</i>	Although this variable is sometimes used with <i>PS1</i> to allow the display of the
9862		current time in the prompt of the user, it is not one that would be manipulated

9863 frequently enough by an interactive user to include in the Shell and Utilities  
 9864 volume of IEEE Std 1003.1-2001.

## 9865 C.2.6 Word Expansions

9866 Step (2) refers to the “portions of fields generated by step (1)”. For example, if the word being  
 9867 expanded were "\$x+\$y" and *IFS*=+, the word would be split only if "\$x" or "\$y" contained  
 9868 '+'; the '+' in the original word was not generated by step (1).

9869 *IFS* is used for performing field splitting on the results of parameter and command substitution;  
 9870 it is not used for splitting all fields. Previous versions of the shell used it for splitting all fields  
 9871 during field splitting, but this has severe problems because the shell can no longer parse its own  
 9872 script. There are also important security implications caused by this behavior. All useful  
 9873 applications of *IFS* use it for parsing input of the *read* utility and for splitting the results of  
 9874 parameter and command substitution.

9875 The rule concerning expansion to a single field requires that if **foo=abc** and **bar=def**, that:

9876 "\$foo" "\$bar"

9877 expands to the single field:

9878 abcdef

9879 The rule concerning empty fields can be illustrated by:

```
9880 $ unset foo
9881 $ set $foo bar ' ' xyz "$foo" abc
9882 $ for i
9883 > do
9884 >     echo "-$i-"
9885 > done
9886 -bar-
9887 --
9888 -xyz-
9889 --
9890 -abc-
```

9891 Step (1) indicates that parameter expansion, command substitution, and arithmetic expansion  
 9892 are all processed simultaneously as they are scanned. For example, the following is valid  
 9893 arithmetic:

```
9894 x=1
9895 echo $(( $(echo 3)+$x ))
```

9896 An early proposal stated that tilde expansion preceded the other steps, but this is not the case in  
 9897 known historical implementations; if it were, and if a referenced home directory contained a '\$'  
 9898 character, expansions would result within the directory name.

### 9899 C.2.6.1 Tilde Expansion

9900 Tilde expansion generally occurs only at the beginning of words, but an exception based on  
 9901 historical practice has been included:

```
9902 PATH=/posix/bin:~dgg/bin
```

9903 This is eligible for tilde expansion because tilde follows a colon and none of the relevant  
 9904 characters is quoted. Consideration was given to prohibiting this behavior because any of the  
 9905 following are reasonable substitutes:

```

9906     PATH=$(printf %s ~karels/bin : ~bostic/bin)
9907     for Dir in ~maat/bin ~srb/bin ...
9908     do
9909         PATH=${PATH:+$PATH:}$Dir
9910     done

```

9911 In the first command, explicit colons are used for each directory. In all cases, the shell performs  
 9912 tilde expansion on each directory because all are separate words to the shell.

9913 Note that expressions in operands such as:

```

9914     make -k mumble LIBDIR=~chet/lib

```

9915 do not qualify as shell variable assignments, and tilde expansion is not performed (unless the  
 9916 command does so itself, which *make* does not).

9917 Because of the requirement that the word is not quoted, the following are not equivalent; only  
 9918 the last causes tilde expansion:

```

9919     \~hlj/    ~h\lj/    ~"hlj"/    ~hlj\ /    ~hlj/

```

9920 In an early proposal, tilde expansion occurred following any unquoted equals sign or colon, but  
 9921 this was removed because of its complexity and to avoid breaking commands such as:

```

9922     rcp hostname:~marc/.profile .

```

9923 A suggestion was made that the special sequence "\$~" should be allowed to force tilde  
 9924 expansion anywhere. Since this is not historical practice, it has been left for future  
 9925 implementations to evaluate. (The description in the Shell and Utilities volume of  
 9926 IEEE Std 1003.1-2001, Section 2.2, Quoting requires that a dollar sign be quoted to represent  
 9927 itself, so the "\$~" combination is already unspecified.)

9928 The results of giving tilde with an unknown login name are undefined because the KornShell  
 9929 "~+" and "~-" constructs make use of this condition, but in general it is an error to give an  
 9930 incorrect login name with tilde. The results of having *HOME* unset are unspecified because some  
 9931 historical shells treat this as an error.

### 9932 C.2.6.2 Parameter Expansion

9933 The rule for finding the closing '}' in "\${...}" is the one used in the KornShell and is  
 9934 upwardly-compatible with the Bourne shell, which does not determine the closing '}' until the  
 9935 word is expanded. The advantage of this is that incomplete expansions, such as:

```

9936     ${foo

```

9937 can be determined during tokenization, rather than during expansion.

9938 The string length and substring capabilities were included because of the demonstrated need for  
 9939 them, based on their usage in other shells, such as C shell and KornShell.

9940 Historical versions of the KornShell have not performed tilde expansion on the word part of  
 9941 parameter expansion; however, it is more consistent to do so.

## 9942 C.2.6.3 Command Substitution

9943 The "\$ ( )" form of command substitution solves a problem of inconsistent behavior when using  
 9944 backquotes. For example:

9945	Command	Output
9946	echo `\\$x`	\\$x
9947	echo `echo `\\$x` `	\$x
9948	echo \$(echo `\\$x`)	\\$x

9949 Additionally, the backquoted syntax has historical restrictions on the contents of the embedded  
 9950 command. While the newer "\$ ( )" form can process any kind of valid embedded script, the  
 9951 backquoted form cannot handle some valid scripts that include backquotes. For example, these  
 9952 otherwise valid embedded scripts do not work in the left column, but do work on the right:

9953	echo `	echo \$(
9954	cat <<\eof	cat <<\eof
9955	a here-doc with `	a here-doc with )
9956	eof	eof
9957	`	)
9958	echo `	echo \$(
9959	echo abc # a comment with `	echo abc # a comment with )
9960	`	)
9961	echo `	echo \$(
9962	echo ` ` `	echo ` ` `
9963	`	)

9964 Because of these inconsistent behaviors, the backquoted variety of command substitution is not  
 9965 recommended for new applications that nest command substitutions or attempt to embed  
 9966 complex scripts.

9967 The KornShell feature:

9968 If *command* is of the form *<word*, *word* is expanded to generate a pathname, and the value of  
 9969 the command substitution is the contents of this file with any trailing <newline>s deleted.

9970 was omitted from the Shell and Utilities volume of IEEE Std 1003.1-2001 because \$(*cat word*) is  
 9971 an appropriate substitute. However, to prevent breaking numerous scripts relying on this  
 9972 feature, it is unspecified to have a script within "\$ ( )" that has only redirections.

9973 The requirement to separate "\$ ( " and ' ( ' when a single subshell is command-substituted is to  
 9974 avoid any ambiguities with arithmetic expansion.

9975 IEEE Std 1003.1-2001/Cor 1-2002, item XCU/TC1/D6/4 is applied, changing the text from: "If a 1  
 9976 command substitution occurs inside double-quotes, it shall not be performed on the results of 1  
 9977 the substitution." to: "If a command substitution occurs inside double-quotes, field splitting and 1  
 9978 pathname expansion shall not be performed on the results of the substitution.". The 1  
 9979 replacement text taken from the ISO POSIX-2:1993 standard is clearer about the items that are 1  
 9980 not performed. 1

## 9981 C.2.6.4 Arithmetic Expansion

9982 The standard developers agreed that there was a strong desire for some kind of arithmetic 2  
 9983 evaluator to provide functionality similar to *expr*, that relating it to '\$' makes it work well with 2  
 9984 the standard shell language and provides access to arithmetic evaluation in places where 2  
 9985 accessing a utility would be inconvenient. 2

9986 The syntax and semantics for arithmetic were revised for the ISO/IEC 9945-2:1993 standard. 2  
 9987 The language represents a simple subset of the previous arithmetic language (which was 2  
 9988 derived from the KornShell "(( ))" construct). The syntax was changed from that of a 2  
 9989 command denoted by ((*expression*)) to an expansion denoted by \$(*expression*). The new form is 2  
 9990 a dollar expansion ('\$') that evaluates the expression and substitutes the resulting value.  
 9991 Objections to the previous style of arithmetic included that it was too complicated, did not fit in  
 9992 well with the use of variables in the shell, and its syntax conflicted with subshells. The  
 9993 justification for the new syntax is that the shell is traditionally a macro language, and if a new  
 9994 feature is to be added, it should be accomplished by extending the capabilities presented by the  
 9995 current model of the shell, rather than by inventing a new one outside the model; adding a new  
 9996 dollar expansion was perceived to be the most intuitive and least destructive way to add such a  
 9997 new capability.

9998 The standard requires assignment operators to be supported (as listed in the Shell and Utilities 2  
 9999 volume of IEEE Std 1003.1-2001, Section 1.7.2, Concepts Derived from the ISO Standard), and 2  
 10000 since arithmetic expansions are not specified to be evaluated in a subshell environment, changes 2  
 10001 to variables there have to be in effect after the arithmetic expansion, just as in the parameter 2  
 10002 expansion "\${x=value}". 2

10003 Note, however, that "\$(( x=5 ))" need not be equivalent to "\$(( \$x=5 ))". If the value of 2  
 10004 the environment variable *x* is the string "y=", the expansion of "\$(( x=5 ))" would set *x* to 5 2  
 10005 and output 5, but "\$(( \$x=5 ))" would output 0 if the value of the environment variable *y* is 2  
 10006 not 5 and would output 1 if the environment variable *y* is 5. Similarly, if the value of the 2  
 10007 environment variable is 4, the expansion of "\$(( x=5 ))" would still set *x* to 5 and output 5, 2  
 10008 but "\$(( \$x=5 ))" (which would be equivalent to "\$(( 4=5 ))") would yield a syntax 2  
 10009 error. 2

10010 In early proposals, a form \$[*expression*] was used. It was functionally equivalent to the "\$(( ))" 2  
 10011 of the current text, but objections were lodged that the 1988 KornShell had already implemented 2  
 10012 "\$(( ))" and there was no compelling reason to invent yet another syntax. Furthermore, the 2  
 10013 "\$[ ]" syntax had a minor incompatibility involving the patterns in **case** statements. 2

10014 The portion of the ISO C standard arithmetic operations selected corresponds to the operations 2  
 10015 historically supported in the KornShell. In addition to the exceptions listed in the Shell and 2  
 10016 Utilities volume of IEEE Std 1003.1-2001, Section 2.6.4, Arithmetic Expansion, the use of the 2  
 10017 following are explicitly outside the scope of the rules defined in the Shell and Utilities volume of 2  
 10018 IEEE Std 1003.1-2001, Section 1.7.2.1, Arithmetic Precision and Operations: 2

- 10019 • The prefix operator '&' and the "[ ]", "->", and '. ' operators. 2
- 10020 • Casts 2

10021 It was concluded that the *test* command (**D**) was sufficient for the majority of relational arithmetic 2  
 10022 tests, and that tests involving complicated relational expressions within the shell are rare, yet 2  
 10023 could still be accommodated by testing the value of "\$(( ))" itself. For example: 2

```
10024 # a complicated relational expression
10025 while [ $(( (($x + $y)/($a * $b)) < ($foo*$bar) )) -ne 0 ]
```

10026 or better yet, the rare script that has many complex relational expressions could define a 2  
 10027 function like this: 2

```

10028     val() {
10029         return $((!$1))
10030     }

```

10031 and complicated tests would be less intimidating:

```

10032     while val $(( ($x + $y)/($a * $b)) < ($foo*$bar) ))
10033     do
10034         # some calculations
10035     done

```

10036 A suggestion that was not adopted was to modify *true* and *false* to take an optional argument,  
 10037 and *true* would exit true only if the argument was non-zero, and *false* would exit false only if the  
 10038 argument was non-zero:

```

10039     while true $(( $x > 5 && $y <= 25 ))

```

10040 There is a minor portability concern with the new syntax. The example "`$((2+2))`" could have  
 10041 been intended to mean a command substitution of a utility named "2+2" in a subshell. The  
 10042 standard developers considered this to be obscure and isolated to some KornShell scripts  
 10043 (because "`$()`" command substitution existed previously only in the KornShell). The text on  
 10044 command substitution requires that the "`$`" and "`(`" be separate tokens if this usage is needed.

10045 An example such as:

```

10046     echo $((echo hi);(echo there))

```

10047 should not be misinterpreted by the shell as arithmetic because attempts to balance the  
 10048 parentheses pairs would indicate that they are subshells. However, as indicated by the Base  
 10049 Definitions volume of IEEE Std 1003.1-2001, Section 3.112, Control Operator, a conforming  
 10050 application must separate two adjacent parentheses with white space to indicate nested  
 10051 subshells.

10052 The standard is intentionally silent about how a variable's numeric value in an expression is 2  
 10053 determined from its normal "sequence of bytes" value. It could be done as a text substitution, as 2  
 10054 a conversion like that performed by *strtol()*, or even recursive evaluation. Therefore, the only 2  
 10055 cases for which the standard is clear are those for which both conversions produce the same 2  
 10056 result. The cases where they give the same result are those where the sequence of bytes form a 2  
 10057 valid integer constant. Therefore, if a variable does not contain a valid integer constant, the 2  
 10058 behavior is unspecified. 2

10059 For the commands: 2

```

10060 x=010; echo $((x += 1)) 2

```

10061 the output must be 9. 2

10062 For the commands: 2

```

10063 x=' 1'; echo $((x += 1)) 2

```

10064 the results are unspecified. 2

10065 For the commands: 2

```

10066 x=1+1; echo $((x += 1)) 2

```

10067 the results are unspecified. 2

10068 Although the ISO/IEC 9899:1999 standard now requires support for **long long** and allows  
 10069 extended integer types with higher ranks, IEEE Std 1003.1-2001 only requires arithmetic  
 10070 expansions to support **signed long** integer arithmetic. Implementations are encouraged to

10071 support signed integer values at least as large as the size of the largest file allowed on the  
10072 implementation.

10073 Implementations are also allowed to perform floating-point evaluations as long as an  
10074 application won't see different results for expressions that would not overflow **signed long**  
10075 integer expression evaluation. (This includes appropriate truncation of results to integer values.)

10076 Changes made in response to IEEE PASC Interpretation 1003.2 #208 removed the requirement  
10077 that the integer constant suffixes `l` and `L` had to be recognized. The ISO POSIX-2: 1993 standard  
10078 did not require the `u`, `ul`, `uL`, `U`, `Ul`, `UL`, `lu`, `lU`, `Lu`, and `LU` suffixes since only signed integer  
10079 arithmetic was required. Since all arithmetic expressions were treated as handling **signed long**  
10080 integer types anyway, the `l` and `L` suffixes were redundant. No known scripts used them and  
10081 some historic shells did not support them. When the ISO/IEC 9899: 1999 standard was used as  
10082 the basis for the description of arithmetic processing, the `ll` and `LL` suffixes and combinations  
10083 were also not required. Implementations are still free to accept any or all of these suffixes, but  
10084 are not required to do so.

10085 There was also some confusion as to whether the shell was required to recognize character  
10086 constants. Syntactically, character constants were required to be recognized, but the  
10087 requirements for the handling of backslash (`'\'`) and quote (`' '`) characters (needed to specify  
10088 character constants) within an arithmetic expansion were ambiguous. Furthermore, no known  
10089 shells supported them. Changes made in response to IEEE PASC Interpretation 1003.2 #208  
10090 removed the requirement to support them (if they were indeed required before).  
10091 IEEE Std 1003.1-2001 clearly does not require support for character constants. 1

10092 IEEE Std 1003.1-2001/Cor 2-2004, item XCU/TC2/D6/3 is applied, clarifying arithmetic 1  
10093 expressions. 1

#### 10094 C.2.6.5 Field Splitting

10095 The operation of field splitting using *IFS*, as described in early proposals, was based on the way  
10096 the KornShell splits words, but it is incompatible with other common versions of the shell.  
10097 However, each has merit, and so a decision was made to allow both. If the *IFS* variable is unset  
10098 or is `<space><tab><newline>`, the operation is equivalent to the way the System V shell splits  
10099 words. Using characters outside the `<space><tab><newline>` set yields the KornShell behavior,  
10100 where each of the non-`<space><tab><newline>`s is significant. This behavior, which affords the  
10101 most flexibility, was taken from the way the original *awk* handled field splitting.

10102 Rule (3) can be summarized as a pseudo-ERE:

10103  $(s^*ns^* | s^+)$

10104 where *s* is an *IFS* white space character and *n* is a character in the *IFS* that is not white space.  
10105 Any string matching that ERE delimits a field, except that the *s*<sup>+</sup> form does not delimit fields at  
10106 the beginning or the end of a line. For example, if *IFS* is `<space>/<comma>/<tab>`, the string:

10107 `<space><space>red<space><space>, <space>white<space>blue`

10108 yields the three colors as the delimited fields.

#### 10109 C.2.6.6 Pathname Expansion

10110 There is no additional rationale provided for this section.



## 10111 C.2.6.7 Quote Removal

10112 There is no additional rationale provided for this section.

## 10113 C.2.7 Redirection

10114 In the System Interfaces volume of IEEE Std 1003.1-2001, file descriptors are integers in the range  
 10115 0–({OPEN\_MAX}–1). The file descriptors discussed in the Shell and Utilities volume of  
 10116 IEEE Std 1003.1-2001, Section 2.7, Redirection are that same set of small integers.

10117 Having multi-digit file descriptor numbers for I/O redirection can cause some obscure  
 10118 compatibility problems. Specifically, scripts that depend on an example command:

10119 `echo 22>/dev/null`

10120 echoing "2" to standard error or "22" to standard output are no longer portable. However, the  
 10121 file descriptor number must still be delimited from the preceding text. For example:

10122 `cat file2>foo`

10123 writes the contents of **file2**, not the contents of **file**.

10124 The ">|" format of output redirection was adopted from the KornShell. Along with the  
 10125 *noclobber* option, *set -C*, it provides a safety feature to prevent inadvertent overwriting of  
 10126 existing files. (See the RATIONALE for the *pathchk* utility for why this step was taken.) The  
 10127 restriction on regular files is historical practice.

10128 The System V shell and the KornShell have differed historically on pathname expansion of *word*;  
 10129 the former never performed it, the latter only when the result was a single field (file). As a  
 10130 compromise, it was decided that the KornShell functionality was useful, but only as a shorthand  
 10131 device for interactive users. No reasonable shell script would be written with a command such  
 10132 as:

10133 `cat foo > a*`

10134 Thus, shell scripts are prohibited from doing it, while interactive users can select the shell with  
 10135 which they are most comfortable.

10136 The construct "2>&1" is often used to redirect standard error to the same file as standard  
 10137 output. Since the redirections take place beginning to end, the order of redirections is significant.  
 10138 For example:

10139 `ls > foo 2>&1`

10140 directs both standard output and standard error to file **foo**. However:

10141 `ls 2>&1 > foo`

10142 only directs standard output to file **foo** because standard error was duplicated as standard  
 10143 output before standard output was directed to file **foo**.

10144 The "<>" operator could be useful in writing an application that worked with several terminals,  
 10145 and occasionally wanted to start up a shell. That shell would in turn be unable to run  
 10146 applications that run from an ordinary controlling terminal unless it could make use of "<>"  
 10147 redirection. The specific example is a historical version of the pager *more*, which reads from  
 10148 standard error to get its commands, so standard input and standard output are both available  
 10149 for their usual usage. There is no way of saying the following in the shell without "<>":

10150 `cat food | more - >/dev/tty03 2<>/dev/tty03`

10151 Another example of "<>" is one that opens **/dev/tty** on file descriptor 3 for reading and writing:

10152            `exec 3<> /dev/tty`

10153            An example of creating a lock file for a critical code region:

```
10154            set -C
10155            until     2> /dev/null > lockfile
10156            do        sleep 30
10157            done
10158            set +C
10159            perform critical function
10160            rm lockfile
```

10161            Since `/dev/null` is not a regular file, no error is generated by redirecting to it in *noclobber* mode.

10162            Tilde expansion is not performed on a here-document because the data is treated as if it were  
10163            enclosed in double quotes.

#### 10164 C.2.7.1 *Redirecting Input*

10165            There is no additional rationale provided for this section.

#### 10166 C.2.7.2 *Redirecting Output*

10167            There is no additional rationale provided for this section.

#### 10168 C.2.7.3 *Appending Redirected Output*

10169            Note that when a file is opened (even with the `O_APPEND` flag set), the initial file offset for that  
10170            file is set to the beginning of the file. Some historic shells set the file offset to the current end-of-  
10171            file when **append** mode shell redirection was used, but this is not allowed by  
10172            IEEE Std 1003.1-2001.

#### 10173 C.2.7.4 *Here-Document*

10174            There is no additional rationale provided for this section.

#### 10175 C.2.7.5 *Duplicating an Input File Descriptor*

10176            There is no additional rationale provided for this section.

#### 10177 C.2.7.6 *Duplicating an Output File Descriptor*

10178            There is no additional rationale provided for this section.

#### 10179 C.2.7.7 *Open File Descriptors for Reading and Writing*

10180            There is no additional rationale provided for this section.

### 10181 C.2.8 **Exit Status and Errors**

#### 10182 C.2.8.1 *Consequences of Shell Errors*

10183            There is no additional rationale provided for this section.

10184 C.2.8.2 *Exit Status for Commands*

10185 There is a historical difference in *sh* and *ksh* non-interactive error behavior. When a command  
 10186 named in a script is not found, some implementations of *sh* exit immediately, but *ksh* continues  
 10187 with the next command. Thus, the Shell and Utilities volume of IEEE Std 1003.1-2001 says that  
 10188 the shell “may” exit in this case. This puts a small burden on the programmer, who has to test  
 10189 for successful completion following a command if it is important that the next command not be  
 10190 executed if the previous command was not found. If it is important for the command to have  
 10191 been found, it was probably also important for it to complete successfully. The test for successful  
 10192 completion would not need to change.

10193 Historically, shells have returned an exit status of  $128+n$ , where  $n$  represents the signal number.  
 10194 Since signal numbers are not standardized, there is no portable way to determine which signal  
 10195 caused the termination. Also, it is possible for a command to exit with a status in the same range  
 10196 of numbers that the shell would use to report that the command was terminated by a signal.  
 10197 Implementations are encouraged to choose exit values greater than 256 to indicate programs  
 10198 that terminate by a signal so that the exit status cannot be confused with an exit status generated  
 10199 by a normal termination.

10200 Historical shells make the distinction between “utility not found” and “utility found but cannot  
 10201 execute” in their error messages. By specifying two seldomly used exit status values for these  
 10202 cases, 127 and 126 respectively, this gives an application the opportunity to make use of this  
 10203 distinction without having to parse an error message that would probably change from locale to  
 10204 locale. The *command*, *env*, *nohup*, and *xargs* utilities in the Shell and Utilities volume of  
 10205 IEEE Std 1003.1-2001 have also been specified to use this convention.

10206 When a command fails during word expansion or redirection, most historical implementations  
 10207 exit with a status of 1. However, there was some sentiment that this value should probably be  
 10208 much higher so that an application could distinguish this case from the more normal exit status  
 10209 values. Thus, the language “greater than zero” was selected to allow either method to be  
 10210 implemented.

10211 C.2.9 **Shell Commands**

10212 A description of an “empty command” was removed from an early proposal because it is only  
 10213 relevant in the cases of *sh -c " "*, *system(" ")*, or an empty shell-script file (such as the  
 10214 implementation of *true* on some historical systems). Since it is no longer mentioned in the Shell  
 10215 and Utilities volume of IEEE Std 1003.1-2001, it falls into the silently unspecified category of  
 10216 behavior where implementations can continue to operate as they have historically, but  
 10217 conforming applications do not construct empty commands. (However, note that *sh* does  
 10218 explicitly state an exit status for an empty string or file.) In an interactive session or a script with  
 10219 other commands, extra <newline>s or semicolons, such as:

```
10220     $ false
10221     $
10222     $ echo $?
10223     1
```

10224 would not qualify as the empty command described here because they would be consumed by  
 10225 other parts of the grammar.

## 10226 C.2.9.1 Simple Commands

10227 The enumerated list is used only when the command is actually going to be executed. For  
 10228 example, in:

10229 `true || $foo *`

10230 no expansions are performed.

10231 The following example illustrates both how a variable assignment without a command name  
 10232 affects the current execution environment, and how an assignment with a command name only  
 10233 affects the execution environment of the command:

```
10234 $ x=red
10235 $ echo $x
10236 red
10237 $ export x
10238 $ sh -c 'echo $x'
10239 red
10240 $ x=blue sh -c 'echo $x'
10241 blue
10242 $ echo $x
10243 red
```

10244 This next example illustrates that redirections without a command name are still performed:

```
10245 $ ls foo
10246 ls: foo: no such file or directory
10247 $ > foo
10248 $ ls foo
10249 foo
```

10250 A command without a command name, but one that includes a command substitution, has an  
 10251 exit status of the last command substitution that the shell performed. For example:

```
10252 if      x=$(command)
10253 then    ...
10254 fi
```

10255 An example of redirections without a command name being performed in a subshell shows that  
 10256 the here-document does not disrupt the standard input of the **while** loop:

```
10257 IFS=:
10258 while read a b
10259 do     echo $a
10260       <<-eof
10261       Hello
10262       eof
10263 done </etc/passwd
```

10264 Following are examples of commands without command names in AND-OR lists:

```

10265     > foo || {
10266         echo "error: foo cannot be created" >&2
10267         exit 1
10268     }

10269     # set saved if /vmunix.save exists
10270     test -f /vmunix.save && saved=1

```

10271 Command substitution and redirections without command names both occur in subshells, but  
 10272 they are not necessarily the same ones. For example, in:

```

10273     exec 3> file
10274     var=$(echo foo >&3) 3>&1

```

10275 it is unspecified whether **foo** is echoed to the file or to standard output.

## 10276 Command Search and Execution

10277 This description requires that the shell can execute shell scripts directly, even if the underlying  
 10278 system does not support the common "#!" interpreter convention. That is, if file **foo** contains  
 10279 shell commands and is executable, the following executes **foo**:

```

10280     ./foo

```

10281 The command search shown here does not match all historical implementations. A more typical  
 10282 sequence has been:

- 10283 • Any built-in (special or regular)
- 10284 • Functions
- 10285 • Path search for executable files

10286 But there are problems with this sequence. Since the programmer has no idea in advance which  
 10287 utilities might have been built into the shell, a function cannot be used to override portably a  
 10288 utility of the same name. (For example, a function named *cd* cannot be written for many  
 10289 historical systems.) Furthermore, the *PATH* variable is partially ineffective in this case, and only  
 10290 a pathname with a slash can be used to ensure a specific executable file is invoked.

10291 After the *execve()* failure described, the shell normally executes the file as a shell script. Some  
 10292 implementations, however, attempt to detect whether the file is actually a script and not an  
 10293 executable from some other architecture. The method used by the KornShell is allowed by the  
 10294 text that indicates non-text files may be bypassed.

10295 The sequence selected for the Shell and Utilities volume of IEEE Std 1003.1-2001 acknowledges  
 10296 that special built-ins cannot be overridden, but gives the programmer full control over which  
 10297 versions of other utilities are executed. It provides a means of suppressing function lookup (via  
 10298 the *command* utility) for the user's own functions and ensures that any regular built-ins or  
 10299 functions provided by the implementation are under the control of the path search. The  
 10300 mechanisms for associating built-ins or functions with executable files in the path are not  
 10301 specified by the Shell and Utilities volume of IEEE Std 1003.1-2001, but the wording requires that  
 10302 if either is implemented, the application is not able to distinguish a function or built-in from an  
 10303 executable (other than in terms of performance, presumably). The implementation ensures that  
 10304 all effects specified by the Shell and Utilities volume of IEEE Std 1003.1-2001 resulting from the  
 10305 invocation of the regular built-in or function (interaction with the environment, variables, traps,  
 10306 and so on) are identical to those resulting from the invocation of an executable file.

10307 IEEE Std 1003.1-2001/Cor 2-2004, item XCU/TC2/D6/4 is applied, updating the case where  
 10308 *execve()* fails due to an error equivalent to the [ENOEXEC] error.

2  
2

**Examples**

Consider three versions of the *ls* utility:

1. The application includes a shell function named *ls*.
2. The user writes a utility named *ls* and puts it in **/fred/bin**.
3. The example implementation provides *ls* as a regular shell built-in that is invoked (either by the shell or directly by *exec*) when the path search reaches the directory **/posix/bin**.

If *PATH*=**/posix/bin**, various invocations yield different versions of *ls*:

Invocation	Version of <i>ls</i>
<i>ls</i> (from within application script)	(1) function
<i>command ls</i> (from within application script)	(3) built-in
<i>ls</i> (from within makefile called by application)	(3) built-in
<i>system("ls")</i>	(3) built-in
<i>PATH="/fred/bin:\$PATH" ls</i>	(2) user's version

**C.2.9.2 Pipelines**

Because pipeline assignment of standard input or standard output or both takes place before redirection, it can be modified by redirection. For example:

```
$ command1 2>&1 | command2
```

sends both the standard output and standard error of *command1* to the standard input of *command2*.

The reserved word **!** allows more flexible testing using AND and OR lists.

It was suggested that it would be better to return a non-zero value if any command in the pipeline terminates with non-zero status (perhaps the bitwise-inclusive OR of all return values). However, the choice of the last-specified command semantics are historical practice and would cause applications to break if changed. An example of historical behavior:

```
$ sleep 5 | (exit 4)
$ echo $?
4
$ (exit 4) | sleep 5
$ echo $?
0
```

**C.2.9.3 Lists**

The equal precedence of **"&&"** and **"||"** is historical practice. The standard developers evaluated the model used more frequently in high-level programming languages, such as C, to allow the shell logical operators to be used for complex expressions in an unambiguous way, but they could not allow historical scripts to break in the subtle way unequal precedence might cause. Some arguments were posed concerning the **"{"** or **"("** groupings that are required historically. There are some disadvantages to these groupings:

- The **"("** can be expensive, as they spawn other processes on some implementations. This performance concern is primarily an implementation issue.
- The **"{"** braces are not operators (they are reserved words) and require a trailing space after each **'{'**, and a semicolon before each **'}'**. Most programmers (and certainly

10351 interactive users) have avoided braces as grouping constructs because of the problematic  
 10352 syntax required. Braces were not changed to operators because that would generate  
 10353 compatibility issues even greater than the precedence question; braces appear outside the  
 10354 context of a keyword in many shell scripts.

10355 IEEE PASC Interpretation 1003.2 #204 is applied, clarifying that the operators "&&" and "||"  
 10356 are evaluated with left associativity.

### 10357 **Asynchronous Lists**

10358 The grammar treats a construct such as:

```
10359     foo & bar & bam &
```

10360 as one “asynchronous list”, but since the status of each element is tracked by the shell, the term  
 10361 “element of an asynchronous list” was introduced to identify just one of the **foo**, **bar**, or **bam**  
 10362 portions of the overall list.

10363 Unless the implementation has an internal limit, such as {CHILD\_MAX}, on the retained process  
 10364 IDs, it would require unbounded memory for the following example:

```
10365     while true
10366     do         foo & echo $!
10367     done
```

10368 The treatment of the signals SIGINT and SIGQUIT with asynchronous lists is described in the  
 10369 Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.11, Signals and Error Handling.

10370 Since the connection of the input to the equivalent of **/dev/null** is considered to occur before  
 10371 redirections, the following script would produce no output:

```
10372     exec < /etc/passwd
10373     cat <&0 &
10374     wait
```

### 10375 **Sequential Lists**

10376 There is no additional rationale provided for this section.

### 10377 **AND Lists**

10378 There is no additional rationale provided for this section.

### 10379 **OR Lists**

10380 There is no additional rationale provided for this section.

## 10381 C.2.9.4 Compound Commands

## 10382 Grouping Commands

10383 The semicolon shown in *{compound-list;}* is an example of a control operator delimiting the }  
 10384 reserved word. Other delimiters are possible, as shown in the Shell and Utilities volume of  
 10385 IEEE Std 1003.1-2001, Section 2.10, Shell Grammar; <newline> is frequently used.

10386 A proposal was made to use the <do-done> construct in all cases where command grouping in  
 10387 the current process environment is performed, identifying it as a construct for the grouping  
 10388 commands, as well as for shell functions. This was not included because the shell already has a  
 10389 grouping construct for this purpose ("{"}), and changing it would have been counter-  
 10390 productive.

## 10391 For Loop

10392 The format is shown with generous usage of <newline>s. See the grammar in the Shell and  
 10393 Utilities volume of IEEE Std 1003.1-2001, Section 2.10, Shell Grammar for a precise description of  
 10394 where <newline>s and semicolons can be interchanged.

10395 Some historical implementations support ' ' and ' ' as substitutes for **do** and **done**. The  
 10396 standard developers chose to omit them, even as an obsolescent feature. (Note that these  
 10397 substitutes were only for the **for** command; the **while** and **until** commands could not use them  
 10398 historically because they are followed by compound-lists that may contain "{...}" grouping  
 10399 commands themselves.)

10400 The reserved word pair **do** ... **done** was selected rather than **do** ... **od** (which would have  
 10401 matched the spirit of **if** ... **fi** and **case** ... **esac**) because *od* is already the name of a standard  
 10402 utility.

10403 PASC Interpretation 1003.2 #169 has been applied changing the grammar.

## 10404 Case Conditional Construct

10405 An optional left parenthesis before *pattern* was added to allow numerous historical KornShell  
 10406 scripts to conform. At one time, using the leading parenthesis was required if the **case** statement  
 10407 was to be embedded within a "\$ ( )" command substitution; this is no longer the case with the  
 10408 POSIX shell. Nevertheless, many historical scripts use the left parenthesis, if only because it  
 10409 makes matching-parenthesis searching easier in *vi* and other editors. This is a relatively simple  
 10410 implementation change that is upwards-compatible for all scripts.

10411 Consideration was given to requiring *break* inside the *compound-list* to prevent falling through to  
 10412 the next pattern action list. This was rejected as being nonexistent practice. An interesting  
 10413 undocumented feature of the KornShell is that using ";" instead of ";;" as a terminator  
 10414 causes the exact opposite behavior—the flow of control continues with the next *compound-list*.

10415 The pattern '\*', given as the last pattern in a **case** construct, is equivalent to the default case in  
 10416 a C-language **switch** statement.

10417 The grammar shows that reserved words can be used as patterns, even if one is the first word on  
 10418 a line. Obviously, the reserved word **esac** cannot be used in this manner.



10419 **If Conditional Construct**

10420 The precise format for the command syntax is described in the Shell and Utilities volume of  
 10421 IEEE Std 1003.1-2001, Section 2.10, Shell Grammar.

10422 **While Loop**

10423 The precise format for the command syntax is described in the Shell and Utilities volume of  
 10424 IEEE Std 1003.1-2001, Section 2.10, Shell Grammar.

10425 **Until Loop**

10426 The precise format for the command syntax is described in the Shell and Utilities volume of  
 10427 IEEE Std 1003.1-2001, Section 2.10, Shell Grammar.

10428 *C.2.9.5 Function Definition Command*

10429 The description of functions in an early proposal was based on the notion that functions should  
 10430 behave like miniature shell scripts; that is, except for sharing variables, most elements of an  
 10431 execution environment should behave as if they were a new execution environment, and  
 10432 changes to these should be local to the function. For example, traps and options should be reset  
 10433 on entry to the function, and any changes to them do not affect the traps or options of the caller.  
 10434 There were numerous objections to this basic idea, and the opponents asserted that functions  
 10435 were intended to be a convenient mechanism for grouping common commands that were to be  
 10436 executed in the current execution environment, similar to the execution of the *dot* special  
 10437 built-in.

10438 It was also pointed out that the functions described in that early proposal did not provide a local  
 10439 scope for everything a new shell script would, such as the current working directory, or *umask*,  
 10440 but instead provided a local scope for only a few select properties. The basic argument was that  
 10441 if a local scope is needed for the execution environment, the mechanism already existed: the  
 10442 application can put the commands in a new shell script and call that script. All historical shells  
 10443 that implemented functions, other than the KornShell, have implemented functions that operate  
 10444 in the current execution environment. Because of this, traps and options have a global scope  
 10445 within a shell script. Local variables within a function were considered and included in another  
 10446 early proposal (controlled by the special built-in *local*), but were removed because they do not fit  
 10447 the simple model developed for functions and because there was some opposition to adding yet  
 10448 another new special built-in that was not part of historical practice. Implementations should  
 10449 reserve the identifier *local* (as well as *typeset*, as used in the KornShell) in case this local variable  
 10450 mechanism is adopted in a future version of IEEE Std 1003.1-2001.

10451 A separate issue from the execution environment of a function is the availability of that function  
 10452 to child shells. A few objectors maintained that just as a variable can be shared with child shells  
 10453 by exporting it, so should a function. In early proposals, the *export* command therefore had a *-f*  
 10454 flag for exporting functions. Functions that were exported were to be put into the environment  
 10455 as *name()=value* pairs, and upon invocation, the shell would scan the environment for these and  
 10456 automatically define these functions. This facility was strongly opposed and was omitted. Some  
 10457 of the arguments against exportable functions were as follows:

- 10458 • There was little historical practice. The Ninth Edition shell provided them, but there was  
 10459 controversy over how well it worked.
- 10460 • There are numerous security problems associated with functions appearing in the  
 10461 environment of a user and overriding standard utilities or the utilities owned by the  
 10462 application.

- There was controversy over requiring *make* to import functions, where it has historically used an *exec* function for many of its command line executions.

- Functions can be big and the environment is of a limited size. (The counter-argument was that functions are no different from variables in terms of size: there can be big ones, and there can be small ones—and just as one does not export huge variables, one does not export huge functions. However, this might not apply to the average shell-function writer, who typically writes much larger functions than variables.)

As far as can be determined, the functions in the Shell and Utilities volume of IEEE Std 1003.1-2001 match those in System V. Earlier versions of the KornShell had two methods of defining functions:

```
function fname { compound-list }
```

and:

```
fname() { compound-list }
```

The latter used the same definition as the Shell and Utilities volume of IEEE Std 1003.1-2001, but differed in semantics, as described previously. The current edition of the KornShell aligns the latter syntax with the Shell and Utilities volume of IEEE Std 1003.1-2001 and keeps the former as is.

The name space for functions is limited to that of a *name* because of historical practice. Complications in defining the syntactic rules for the function definition command and in dealing with known extensions such as the "@()" usage in the KornShell prevented the name space from being widened to a *word*. Using functions to support synonyms such as the "!!" and '% ' usage in the C shell is thus disallowed to conforming applications, but acceptable as an extension. For interactive users, the aliasing facilities in the Shell and Utilities volume of IEEE Std 1003.1-2001 should be adequate for this purpose. It is recognized that the name space for utilities in the file system is wider than that currently supported for functions, if the portable filename character set guidelines are ignored, but it did not seem useful to mandate extensions in systems for so little benefit to conforming applications.

The "()" in the function definition command consists of two operators. Therefore, intermixing <blank>s with the *fname*, '(', and ')' is allowed, but unnecessary.

An example of how a function definition can be used wherever a simple command is allowed:

```
# If variable i is equal to "yes",
# define function foo to be ls -l
#
[ "$i" = yes ] && foo() {
    ls -l
}
```

## C.2.10 Shell Grammar

There are several subtle aspects of this grammar where conventional usage implies rules about the grammar that in fact are not true.

For *compound\_list*, only the forms that end in a *separator* allow a reserved word to be recognized, so usually only a *separator* can be used where a compound list precedes a reserved word (such as **Then**, **Else**, **Do**, and **Rbrace**). Explicitly requiring a separator would disallow such valid (if rare) statements as:

```
if (false) then (echo x) else (echo y) fi
```

- 10507 See the Note under special grammar rule (1).
- 10508 Concerning the third sentence of rule (1) (“Also, if the parser ...”):
- 10509 • This sentence applies rather narrowly: when a compound list is terminated by some clear  
10510 delimiter (such as the closing **fi** of an inner **if\_clause**) then it would apply; where the  
10511 compound list might continue (as in after a ‘;’), rule (7a) (and consequently the first  
10512 sentence of rule (1)) would apply. In many instances the two conditions are identical, but this  
10513 part of rule (1) does not give license to treating a **WORD** as a reserved word unless it is in a  
10514 place where a reserved word has to appear.
  - 10515 • The statement is equivalent to requiring that when the LR(1) lookahead set contains exactly  
10516 one reserved word, it must be recognized if it is present. (Here “LR(1)” refers to the  
10517 theoretical concepts, not to any real parser generator.)
- 10518 For example, in the construct below, and when the parser is at the point marked with ‘^’,  
10519 the only next legal token is **then** (this follows directly from the grammar rules):
- ```
10520     if if...fi then ... fi
10521         ^
```
- 10522 At that point, the **then** must be recognized as a reserved word.
- 10523 (Depending on the parser generator actually used, “extra” reserved words may be in some  
10524 lookahead sets. It does not really matter if they are recognized, or even if any possible  
10525 reserved word is recognized in that state, because if it is recognized and is not in the  
10526 (theoretical) LR(1) lookahead set, an error is ultimately detected. In the example above, if  
10527 some other reserved word (for example, **while**) is also recognized, an error occurs later.
- 10528 This is approximately equivalent to saying that reserved words are recognized after other  
10529 reserved words (because it is after a reserved word that this condition occurs), but avoids the  
10530 “except for ...” list that would be required for **case**, **for**, and so on. (Reserved words are of  
10531 course recognized anywhere a *simple\_command* can appear, as well. Other rules take care of  
10532 the special cases of non-recognition, such as rule (4) for **case** statements.)
- 10533 Note that the body of here-documents are handled by token recognition (see the Shell and  
10534 Utilities volume of IEEE Std 1003.1-2001, Section 2.3, Token Recognition) and do not appear in  
10535 the grammar directly. (However, the here-document I/O redirection operator is handled as part  
10536 of the grammar.)
- 10537 The start symbol of the grammar (**complete\_command**) represents either input from the  
10538 command line or a shell script. It is repeatedly applied by the interpreter to its input and  
10539 represents a single “chunk” of that input as seen by the interpreter.

#### 10540 C.2.10.1 Shell Grammar Lexical Conventions

10541 There is no additional rationale provided for this section.

#### 10542 C.2.10.2 Shell Grammar Rules

10543 There is no additional rationale provided for this section.

10544 **C.2.11 Signals and Error Handling**

10545 There is no additional rationale provided for this section.

10546 **C.2.12 Shell Execution Environment**

10547 Some implementations have implemented the last stage of a pipeline in the current environment  
10548 so that commands such as:

10549 `command | read foo`

10550 set variable **foo** in the current environment. This extension is allowed, but not required;  
10551 therefore, a shell programmer should consider a pipeline to be in a subshell environment, but  
10552 not depend on it.

10553 In early proposals, the description of execution environment failed to mention that each  
10554 command in a multiple command pipeline could be in a subshell execution environment. For  
10555 compatibility with some historical shells, the wording was phrased to allow an implementation  
10556 to place any or all commands of a pipeline in the current environment. However, this means that  
10557 a POSIX application must assume each command is in a subshell environment, but not depend  
10558 on it.

10559 The wording about shell scripts is meant to convey the fact that describing “trap actions” can  
10560 only be understood in the context of the shell command language. Outside of this context, such  
10561 as in a C-language program, signals are the operative condition, not traps.

10562 **C.2.13 Pattern Matching Notation**

10563 Pattern matching is a simpler concept and has a simpler syntax than REs, as the former is  
10564 generally used for the manipulation of filenames, which are relatively simple collections of  
10565 characters, while the latter is generally used to manipulate arbitrary text strings of potentially  
10566 greater complexity. However, some of the basic concepts are the same, so this section points  
10567 liberally to the detailed descriptions in the Base Definitions volume of IEEE Std 1003.1-2001,  
10568 Chapter 9, Regular Expressions.

10569 **C.2.13.1 Patterns Matching a Single Character**

10570 Both quoting and escaping are described here because pattern matching must work in three  
10571 separate circumstances:

- 10572 1. Calling directly upon the shell, such as in pathname expansion or in a **case** statement. All  
10573 of the following match the string or file **abc**:

10574 `abc "abc" a"b" c a\bc a[b]c a["b"]c a[\b]c a["\b"]c a?c a*c`

10575 The following do not:

10576 `"a?c" a*c a\[b]c`

- 10577 2. Calling a utility or function without going through a shell, as described for *find* and the  
10578 *fnmatch()* function defined in the System Interfaces volume of IEEE Std 1003.1-2001.

- 10579 3. Calling utilities such as *find*, *cpio*, *tar*, or *pax* through the shell command line. In this case,  
10580 shell quote removal is performed before the utility sees the argument. For example, in:

10581 `find /bin -name "e\c[\h]o" -print`

10582 after quote removal, the backslashes are presented to *find* and it treats them as escape  
10583 characters. Both precede ordinary characters, so the *c* and *h* represent themselves and *echo*  
10584 would be found on many historical systems (that have it in **/bin**). To find a filename that

10585 contained shell special characters or pattern characters, both quoting and escaping are  
10586 required, such as:

10587 `pax -r ... "*a\(\?"`

10588 to extract a filename ending with "a(?".

10589 Conforming applications are required to quote or escape the shell special characters (sometimes  
10590 called metacharacters). If used without this protection, syntax errors can result or  
10591 implementation extensions can be triggered. For example, the KornShell supports a series of  
10592 extensions based on parentheses in patterns.

10593 The restriction on a circumflex in a bracket expression is to allow implementations that support  
10594 pattern matching using the circumflex as the negation character in addition to the exclamation  
10595 mark. A conforming application must use something like "[\^!]" to match either character.

### 10596 C.2.13.2 Patterns Matching Multiple Characters

10597 Since each asterisk matches zero or more occurrences, the patterns "a\*b" and "a\*\*b" have  
10598 identical functionality.

#### 10599 Examples

10600 `a[bc]` Matches the strings "ab" and "ac".

10601 `a*d` Matches the strings "ad", "abd", and "abcd", but not the string "abc".

10602 `a*d*` Matches the strings "ad", "abcd", "abcdef", "aaaad", and "adddd".

10603 `*a*d` Matches the strings "ad", "abcd", "efabcd", "aaaad", and "adddd".

### 10604 C.2.13.3 Patterns Used for Filename Expansion

10605 The caveat about a slash within a bracket expression is derived from historical practice. The  
10606 pattern "a[b/c]d" does not match such pathnames as **abd** or **a/d**. On some implementations  
10607 (including those conforming to the Single UNIX Specification), it matched a pathname of  
10608 literally "a[b/c]d". On other systems, it produced an undefined condition (an unescaped '['  
10609 used outside a bracket expression). In this version, the XSI behavior is now required.

10610 Filenames beginning with a period historically have been specially protected from view on  
10611 UNIX systems. A proposal to allow an explicit period in a bracket expression to match a leading  
10612 period was considered; it is allowed as an implementation extension, but a conforming  
10613 application cannot make use of it. If this extension becomes popular in the future, it will be  
10614 considered for a future version of the Shell and Utilities volume of IEEE Std 1003.1-2001.

10615 Historical systems have varied in their permissions requirements. To match **f\*/bar** has required  
10616 read permissions on the **f\*** directories in the System V shell, but the Shell and Utilities volume of  
10617 IEEE Std 1003.1-2001, the C shell, and KornShell require only search permissions.

10618 **C.2.14 Special Built-In Utilities**

10619 See the RATIONALE sections on the individual reference pages.

10620 **C.3 Batch Environment Services and Utilities**10621 **Scope of the Batch Environment Services and Utilities Option** 1

10622 This section summarizes the deliberations of the IEEE P1003.15 (Batch Environment) working 1  
 10623 group in the development of the Batch Environment Services and Utilities option, which covers 1  
 10624 a set of services and utilities defining a batch processing system. 1

10625 This informative section contains historical information concerning the contents of the  
 10626 amendment and describes why features were included or discarded by the working group.

10627 **History of Batch Systems**

10628 The supercomputing technical committee began as a “Birds Of a Feather” (BOF) at the January  
 10629 1987 Usenix meeting. There was enough general interest to form a supercomputing attachment  
 10630 to the /usr/group working groups. Several subgroups rapidly formed. Of those subgroups, the  
 10631 batch group was the most ambitious. The first early meetings were spent evaluating user needs  
 10632 and existing batch implementations.

10633 To evaluate user needs, individuals from the supercomputing community came and presented  
 10634 their needs. Common requests were flexibility, interoperability, control of resources, and ease-  
 10635 of-use. Backward-compatibility was not an issue. The working group then evaluated some  
 10636 existing systems. The following different systems were evaluated:

- 10637 • PROD
- 10638 • Convex Distributed Batch
- 10639 • NQS
- 10640 • CTSS
- 10641 • MDQS from Ballistics Research Laboratory (BRL)

10642 Finally, NQS was chosen as a model because it satisfied not only the most user requirements, but  
 10643 because it was public domain, already implemented on a variety of hardware platforms, and  
 10644 network-based.

10645 **Historical Implementations of Batch Systems**

10646 Deferred processing of work under the control of a scheduler has been a feature of most  
 10647 proprietary operating systems from the earliest days of multi-user systems in order to maximize  
 10648 utilization of the computer.

10649 The arrival of UNIX systems proved to be a dilemma to many hardware providers and users  
 10650 because it did not include the sophisticated batch facilities offered by the proprietary systems.  
 10651 This omission was rectified in 1986 by NASA Ames Research Center who developed the  
 10652 Network Queuing System (NQS) as a portable UNIX application that allowed the routing and  
 10653 processing of batch “jobs” in a network. To encourage its usage, the product was later put into  
 10654 the public domain. It was promptly picked up by UNIX hardware providers, and ported and  
 10655 developed for their respective hardware and UNIX implementations.

Many major vendors, who traditionally offer a batch-dominated environment, ported the public-domain product to their systems, customized it to support the capabilities of their systems, and added many customer-requested features.

Due to the strong hardware provider and customer acceptance of NQS, it was decided to use NQS as the basis for the POSIX Batch Environment amendment in 1987. Other batch systems considered at the time included CTSS, MDQS (a forerunner of NQS from the Ballistics Research Laboratory), and PROD (a Los Alamos Labs development). None were thought to have both the functionality and acceptability of NQS.

#### NQS Differences from the *at* utility

The base standard *at* and *batch* utilities are not sufficient to meet the batch processing needs in a supercomputing environment and additional functionality in the areas of resource management, job scheduling, system management, and control of output is required.

#### Batch Environment Services and Utilities Option Definitions

1

The concept of a batch job is closely related to a session with a session leader. The main difference is that a batch job does not have a controlling terminal. There has been much debate over whether to use the term “request” or “job”. Job was the final choice because of the historical use of this term in the batch environment.

1

The current definition for job identifiers is not sufficient with the model of destinations. The current definition is:

```
sequence_number.originating_host
```

Using the model of destination, a host may include multiple batch nodes, the location of which is identified uniquely by a name or directory service. If the current definition is used, batch nodes running on the same host would have to coordinate their use of sequence numbers, as sequence numbers are assigned by the originating host. The alternative is to use the originating batch node name instead of the originating host name.

The reasons for wishing to run more than one batch system per host could be the following.

A test and production batch system are maintained on a single host. This is most likely in a development facility, but could also arise when a site is moving from one version to another. The new batch system could be installed as a test version that is completely separate from the production batch system, so that problems can be isolated to the test system. Requiring the batch nodes to coordinate their use of sequence numbers creates a dependency between the two nodes, and that defeats the purpose of running two nodes.

A site has multiple departments using a single host, with different management policies. An example of contention might be in job selection algorithms. One group might want a FIFO type of selection, while another group wishes to use a more complex algorithm based on resource availability. Again, requiring the batch nodes to coordinate is an unnecessary binding.

The proposal eventually accepted was to replace originating host with originating batch node. This supplies sufficient granularity to ensure unique job identifiers. If more than one batch node is on a particular host, they each have their own unique name.

The queue portion of a destination is not part of the job identifier as these are not required to be unique between batch nodes. For instance, two batch nodes may both have queues called small, medium, and large. It is only the batch node name that is uniquely identifiable throughout the batch system. The queue name has no additional function in this context.

Assume there are three batch nodes, each of which has its own name server. On batch node one, there are no queues. On batch node two, there are fifty queues. On batch node three, there are forty queues. The system administrator for batch node one does not have to configure queues, because there are none implemented. However, if a user wishes to send a job to either batch node two or three, the system administrator for batch node one must configure a destination that maps to the appropriate batch node and queue. If every queue is to be made accessible from batch node one, the system administrator has to configure ninety destinations.

To avoid requiring this, there should be a mechanism to allow a user to separate the destination into a batch node name and a queue name. Then, an implementation that is configured to get to all the batch nodes does not need any more configuration to allow a user to get to all of the queues on all of the batch nodes. The node name is used to locate the batch node, while the queue name is sent unchanged to that batch node.

The following are requirements that a destination identifier must be capable of providing:

- The ability to direct a job to a queue in a particular batch node.
- The ability to direct a job to a particular batch node.
- The ability to group at a higher level than just one queue. This includes grouping similar queues across multiple batch nodes (this is a pipe queue).
- The ability to group batch nodes. This allows a user to submit a job to a group name with no knowledge of the batch node configuration. This also provides aliasing as a special case. Aliasing is a group containing only one batch node name. The group name is the alias.

In addition, the administrator has the following requirements:

- The ability to control access to the queues.
- The ability to control access to the batch nodes.
- The ability to control access to groups of queues (pipe queues).
- The ability to configure retry time intervals and durations.

The requirements of the user are met by destination as explained in the following.

The user has the ability to specify a queue name, which is known only to the batch node specified. There is no configuration of these queues required on the submitting node.

The user has the ability to specify a batch node whose name is network-unique. The configuration required is that the batch node be defined as an application, just as other applications such as FTP are configured.

Once a job reaches a queue, it can again become a user of the batch system. The batch node can choose to send the job to another batch node or queue or both. In other words, the routing is at an application level, and it is up to the batch system to choose where the job will be sent. Configuration is up to the batch node where the queue resides. This provides grouping of queues across batch nodes or within a batch node. The user submits the job to a queue, which by definition routes the job to other queues or nodes or both.

A node name may be given to a naming service, which returns multiple addresses as opposed to just one. This provides grouping at a batch node level. This is a local issue, meaning that the batch node must choose only one of these addresses. The list of addresses is not sent with the job, and once the job is accepted on another node, there is no connection between the list and the job. The requirements of the administrator are met by destination as explained in the following.

The control of queues is a batch system issue, and will be done using the batch administrative utilities.



10743 The control of nodes is a network issue, and will be done through whatever network facilities  
10744 are available.

10745 The control of access to groups of queues (pipe queues) is covered by the control of any other  
10746 queue. The fact that the job may then be sent to another destination is not relevant.

10747 The propagation of a job across more than one point-to-point connection was dropped because  
10748 of its complexity and because all of the issues arising from this capability could not be resolved.  
10749 It could be provided as additional functionality at some time in the future.

10750 The addition of *network* as a defined term was done to clarify the difference between a network  
10751 of batch nodes as opposed to a network of hosts. A network of batch nodes is referred to as a  
10752 batch system. The network refers to the actual host configuration. A single host may have  
10753 multiple batch nodes.

10754 In the absence of a standard network naming convention, this option establishes its own  
10755 convention for the sake of consistency and expediency. This is subject to change, should a future  
10756 working group develop a standard naming convention for network pathnames.

### 10757 C.3.1 Batch General Concepts

10758 During the development of the Batch Environment Services and Utilities option, a number of 1  
10759 topics were discussed at length which influenced the wording of the normative text but could 1  
10760 not be included in the final text. The following items are some of the most significant terms and 1  
10761 concepts of those discussed: 1

- 10762 • Small and Consistent Command Set

10763 Often, conventional utilities from UNIX systems have a very complicated utility syntax and  
10764 usage. This can often result in confusion and errors when trying to use them. The Batch 1  
10765 Environment Services and Utilities option utility set, on the other hand, has been paired to a 1  
10766 small set of robust utilities with an orthogonal calling sequence. 1

- 10767 • Checkpoint/Restart

10768 This feature permits an already executing process to checkpoint or save its contents. Some  
10769 implementations permit this at both the batch utility level (for example, checkpointing this  
10770 job upon its abnormal termination) or from within the job itself via a system call. Support of  
10771 checkpoint/restart is optional. A conscious, careful effort was made to make the *qsub* utility  
10772 consistently refer to checkpoint/restart as optional functionality.

- 10773 • Rerunability

10774 When a user submits a job for batch processing, they can designate it “rerunnable” in that it  
10775 will automatically resume execution from the start of the job if the machine on which it was  
10776 executing crashes for some reason. The decision on whether the job will be rerun or not is  
10777 entirely up to the submitter of the job and no decisions will be made within the batch system.  
10778 A job that is rerunnable and has been submitted with the proper checkpoint/restart switch  
10779 will first be checkpointed and execution begun from that point. Furthermore, use of the  
10780 implementation-defined checkpoint/restart feature will not be defined in this context.

- 10781 • Error Codes

10782 All utilities exit with error status zero (0) if successful, one (1) if a user error occurred, and  
10783 two (2) for an internal Batch Environment Services and Utilities option error. 1

- 10784 • Level of Portability

10785 Portability is specified at both the user, operator, and administrator levels. A conforming  
10786 batch implementation prevents identical functionality and behavior at all these levels.

|       |                                                                                                  |   |
|-------|--------------------------------------------------------------------------------------------------|---|
| 10787 | Additionally, portable batch shell scripts with embedded Batch Environment Services and          | 1 |
| 10788 | Utilities option utilities add an additional level of portability.                               | 1 |
| 10789 | • Resource Specification                                                                         |   |
| 10790 | A small set of globally understood resources, such as memory and CPU time, is specified. All     |   |
| 10791 | conforming batch implementations are able to process them in a manner consistent with the        |   |
| 10792 | yet-to-be-developed resource management model. Resources not in this amendment set are           |   |
| 10793 | ignored and passed along as part of the argument stream of the utility.                          |   |
| 10794 | • Queue Position                                                                                 |   |
| 10795 | Queue position is the place a job occupies in a queue. It is dependent on a variety of factors   |   |
| 10796 | such as submission time and priority. Since priority may be affected by the implementation       |   |
| 10797 | of fair share scheduling, the definition of queue position is implementation-defined.            |   |
| 10798 | • Queue ID                                                                                       |   |
| 10799 | A numerical queue ID is an external requirement for purposes of accounting. The                  |   |
| 10800 | identification number was chosen over queue name for processing convenience.                     |   |
| 10801 | • Job ID                                                                                         |   |
| 10802 | A common notion of “jobs” is a collection of processes whose process group cannot be             |   |
| 10803 | altered and is used for resource management and accounting. This concept is                      |   |
| 10804 | implementation-defined and, as such, has been omitted from the batch amendment.                  |   |
| 10805 | • Bytes <i>versus</i> Words                                                                      |   |
| 10806 | Except for one case, bytes are used as the standard unit for memory size. Furthermore, the       |   |
| 10807 | definition of a word varies from machine to machine. Therefore, bytes will be the default unit   |   |
| 10808 | of memory size.                                                                                  |   |
| 10809 | • Regular Expressions                                                                            |   |
| 10810 | The standard definition of regular expressions is much too broad to be used in the batch         |   |
| 10811 | utility syntax. All that is needed is a simple concept of “all”; for example, delete all my jobs |   |
| 10812 | from the named queue. For this reason, regular expressions have been eliminated from the         |   |
| 10813 | batch amendment.                                                                                 |   |
| 10814 | • Display Privacy                                                                                |   |
| 10815 | How much data should be displayed locally through functions? Local policy dictates the           |   |
| 10816 | amount of privacy. Library functions must be used to create and enforce local policy.            |   |
| 10817 | Network and local <i>qstats</i> must reflect the policy of the server machine.                   |   |
| 10818 | • Remote Host Naming Convention                                                                  |   |
| 10819 | It was decided that host names would be a maximum of 255 characters in length, with at           |   |
| 10820 | most 15 characters being shown in displays. The 255 character limit was chosen because it is     |   |
| 10821 | consistent with BSD. The 15-character limit was an arbitrary decision.                           |   |
| 10822 | • Network Administration                                                                         |   |
| 10823 | Network administration is important, but is outside the scope of the batch amendment.            |   |
| 10824 | Network administration could be done with <i>rsh</i> . However, authentication becomes two-      |   |
| 10825 | sided.                                                                                           |   |
| 10826 | • Network Administration Philosophy                                                              |   |
| 10827 | Keep it simple. Centralized management should be possible. For example, Los Alamos needs         |   |
| 10828 | a dumb set of CPUs to be managed by a central system <i>versus</i> several independently-        |   |

- 10829 managed systems as is the general case for the Batch Environment Services and Utilities 1  
 10830 option. 1
- 10831 • Operator Utility Defaults (that is, Default Host, User, Account, and so on)
- 10832 It was decided that usability would override orthogonality and syntactic consistency.
- 10833 • The Batch System Manager and Operator Distinction
- 10834 The distinction between manager and operator is that operators can only control the flow of  
 10835 jobs. A manager can alter the batch system configuration in addition to job flow. POSIX  
 10836 makes a distinction between user and system administrator but goes no further. The  
 10837 concepts of manager and operator privileges fall under local policy. The distinction between  
 10838 manager and operator is historical in batch environments, and the Batch Environment 1  
 10839 Services and Utilities option has continued that distinction. 1
- 10840 • The Batch System Administrator
- 10841 An administrator is equivalent to a batch system manager.

### 10842 C.3.2 Batch Services

- 10843 This rationale is provided as informative rather than normative text, to avoid placing  
 10844 requirements on implementors regarding the use of symbolic constants, but at the same time to  
 10845 give implementors a preferred practice for assigning values to these constants to promote  
 10846 interoperability.
- 10847 The *Checkpoint* and *Minimum\_Cpu\_Interval* attributes induce a variety of behavior depending  
 10848 upon their values. Some jobs cannot or should not be checkpointed. Other users will simply  
 10849 need to ensure job continuation across planned downtimes; for example, scheduled preventive  
 10850 maintenance. For users consuming expensive resources, or for jobs that run longer than the  
 10851 mean time between failures, however, periodic checkpointing may be essential. However,  
 10852 system administrators must be able to set minimum checkpoint intervals on a queue-by-queue  
 10853 basis to guard against, for example, naive users specifying interval values too small on  
 10854 memory-intensive jobs. Otherwise, system overhead would adversely affect performance.
- 10855 The use of symbolic constants, such as NO\_CHECKPOINT, was introduced to lend a degree of  
 10856 formalism and portability to this option.
- 10857 Support for checkpointing is optional for servers. However, clients must provide for the `-c`  
 10858 option, since in a distributed environment the job may run on a server that does provide such  
 10859 support, even if the host of the client does not support the checkpoint feature.
- 10860 If the user does not specify the `-c` option, the default action is left unspecified by this option.  
 10861 Some implementations may wish to do checkpointing by default; others may wish to checkpoint  
 10862 only under an explicit request from the user.
- 10863 The *Priority* attribute has been made non-optional. All clients already had been required to  
 10864 support the `-p` option. The concept of prioritization is common in historical implementations.  
 10865 The default priority is left to the server to establish.
- 10866 The *Hold\_Types* attribute has been modified to allow for implementation-defined hold types to  
 10867 be passed to a batch server.
- 10868 It was the intent of the IEEE P1003.15 working group to mandate the support for the  
 10869 *Resource\_List* attribute in this option by referring to another amendment, specifically the  
 10870 IEEE P1003.1a draft standard. However, during the development of the IEEE P1003.1a draft  
 10871 standard this was excluded. As such this requirement has been removed from the normative  
 10872 text.

10873 The *Shell\_Path* attribute has been modified to accept a list of shell paths that are associated with  
 10874 a host. The name of the attribute has been changed to *Shell\_Path\_List*.

### 10875 C.3.3 Common Behavior for Batch Environment Utilities

10876 This section was defined to meet the goal of a “Small and Consistent Command Set” for this  
 10877 option.

## 10878 C.4 Utilities

10879 For the utilities included in IEEE Std 1003.1-2001, see the RATIONALE sections on the individual  
 10880 reference pages.

### 10881 Exclusion of Utilities

10882 The set of utilities contained in IEEE Std 1003.1-2001 is drawn from the base documents, with  
 10883 one addition: the *c99* utility. This section contains rationale for some of the deliberations that led  
 10884 to this set of utilities, and why certain utilities were excluded.

10885 Many utilities were evaluated by the standard developers; more historical utilities were  
 10886 excluded from the base documents than included. The following list contains many common  
 10887 UNIX system utilities that were not included as mandatory utilities, in the User Portability  
 10888 Utilities option, in the XSI extension, or in one of the software development groups. It is  
 10889 logistically difficult for this rationale to distribute correctly the reasons for not including a utility  
 10890 among the various utility options. Therefore, this section covers the reasons for all utilities not  
 10891 included in IEEE Std 1003.1-2001.

10892 This rationale is limited to a discussion of only those utilities actively or indirectly evaluated by  
 10893 the standard developers of the base documents, rather than the list of all known UNIX utilities  
 10894 from all its variants.

10895 *adb* The intent of the various software development utilities was to assist in the  
 10896 installation (rather than the actual development and debugging) of applications.  
 10897 This utility is primarily a debugging tool. Furthermore, many useful aspects of *adb*  
 10898 are very hardware-specific.

10899 *as* Assemblers are hardware-specific and are included implicitly as part of the  
 10900 compilers in IEEE Std 1003.1-2001.

10901 *banner* The only known use of this command is as part of the *lp* printer header pages. It  
 10902 was decided that the format of the header is implementation-defined, so this utility  
 10903 is superfluous to application portability.

10904 *calendar* This reminder service program is not useful to conforming applications.

10905 *cancel* The *lp* (line printer spooling) system specified is the most basic possible and did  
 10906 not need this level of application control.

10907 *chroot* This is primarily of administrative use, requiring superuser privileges.

10908 *col* No utilities defined in IEEE Std 1003.1-2001 produce output requiring such a filter.  
 10909 The *nroff* text formatter is present on many historical systems and will continue to  
 10910 remain as an extension; *col* is expected to be shipped by all the systems that ship  
 10911 *nroff*.

10912 *cpio* This has been replaced by *pax*, for reasons explained in the rationale for that utility.

|       |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-------|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 10913 | <i>c++</i>    | This is subsumed by <i>c99</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| 10914 | <i>cu</i>     | This utility is terminal-oriented and is not useful from shell scripts or typical application programs.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| 10915 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10916 | <i>dc</i>     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10917 |               | The functionality of this utility can be provided by the <i>bc</i> utility; <i>bc</i> was selected because it was easier to use and had superior functionality. Although the historical versions of <i>bc</i> are implemented using <i>dc</i> as a base, IEEE Std 1003.1-2001 prescribes the interface and not the underlying mechanism used to implement it.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| 10918 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10919 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10920 | <i>dircmp</i> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10921 |               | Although a useful concept, the historical output of this directory comparison program is not suitable for processing in application programs. Also, the <i>diff -r</i> command gives equivalent functionality.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| 10922 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10923 | <i>dis</i>    | Disassemblers are hardware-specific.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| 10924 | <i>emacs</i>  | The community of <i>emacs</i> editing enthusiasts was adamant that the full <i>emacs</i> editor not be included in the base documents because they were concerned that an attempt to standardize this very powerful environment would encourage vendors to ship versions conforming strictly to the standard, but lacking the extensibility required by the community. The author of the original <i>emacs</i> program also expressed his desire to omit the program. Furthermore, there were a number of historical UNIX systems that did not include <i>emacs</i> , or included it without supporting it, but there were very few that did not include and support <i>vi</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| 10925 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10926 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10927 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10928 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10929 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10930 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10931 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10932 | <i>ld</i>     | This is subsumed by <i>c99</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| 10933 | <i>line</i>   | The functionality of <i>line</i> can be provided with <i>read</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10934 | <i>lint</i>   | This technology is partially subsumed by <i>c99</i> . It is also hard to specify the degree of checking for possible error conditions in programs in any compiler, and specifying what <i>lint</i> would do in these cases is equally difficult.<br><br>It is fairly easy to specify what a compiler does. It requires specifying the language, what it does with that language, and stating that the interpretation of any incorrect program is unspecified. Unfortunately, any description of <i>lint</i> is required to specify what to do with erroneous programs. Since the number of possible errors and questionable programming practices is infinite, one cannot require <i>lint</i> to detect all errors of any given class.<br><br>Additionally, some vendors complained that since many compilers are distributed in a binary form without a <i>lint</i> facility (because the ISO C standard does not require one), implementing the standard as a stand-alone product will be much harder. Rather than being able to build upon a standard compiler component (simply by providing <i>c99</i> as an interface), source to that compiler would most likely need to be modified to provide the <i>lint</i> functionality. This was considered a major burden on system providers for a very small gain to developers (users). |
| 10935 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10936 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10937 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10938 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10939 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10940 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10941 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10942 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10943 |               | This utility is terminal-oriented and is not useful from shell scripts or typical application programs.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| 10944 | <i>login</i>  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10945 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10946 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10947 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10948 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10949 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10950 | <i>lorder</i> | This utility is an aid in creating an implementation-defined detail of object libraries that the standard developers did not feel required standardization.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| 10951 |               | The <i>lp</i> system specified is the most basic possible and did not need this level of application control.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| 10952 | <i>lpstat</i> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10953 |               | This utility was omitted in favor of <i>mailx</i> because there was a considerable functionality overlap between the two.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 10954 | <i>mail</i>   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10955 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10956 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 10957 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |

|       |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|-------|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 10958 | <i>mknod</i>  | This was omitted in favor of <i>mkfifo</i> , as <i>mknod</i> has too many implementation-defined functions.                                                                                                                                                                                                                                                                                                                                                           |
| 10959 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10960 | <i>news</i>   | This utility is terminal-oriented and is not useful from shell scripts or typical application programs.                                                                                                                                                                                                                                                                                                                                                               |
| 10961 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10962 | <i>pack</i>   | This compression program was considered inferior to <i>compress</i> .                                                                                                                                                                                                                                                                                                                                                                                                 |
| 10963 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10964 | <i>passwd</i> | This utility was proposed in a historical draft of the base documents but met with too many objections to be included. There were various reasons:                                                                                                                                                                                                                                                                                                                    |
| 10965 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10966 |               | <ul style="list-style-type: none"> <li>• Changing a password should not be viewed as a command, but as part of the login sequence. Changing a password should only be done while a trusted path is in effect.</li> </ul>                                                                                                                                                                                                                                              |
| 10967 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10968 |               | <ul style="list-style-type: none"> <li>• Even though the text in early drafts was intended to allow a variety of implementations to conform, the security policy for one site may differ from another site running with identical hardware and software. One site might use password authentication while the other did not. Vendors could not supply a <i>passwd</i> utility that would conform to IEEE Std 1003.1-2001 for all sites using their system.</li> </ul> |
| 10969 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10970 |               | <ul style="list-style-type: none"> <li>• This is really a subject for a system administration working group or a security working group.</li> </ul>                                                                                                                                                                                                                                                                                                                   |
| 10971 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10972 |               | This compression program was considered inferior to <i>zcat</i> .                                                                                                                                                                                                                                                                                                                                                                                                     |
| 10973 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10974 | <i>pcat</i>   | This duplicated many of the features of the <i>more</i> pager, which was preferred by the standard developers.                                                                                                                                                                                                                                                                                                                                                        |
| 10975 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10976 | <i>pg</i>     | The intent of the various software development utilities was to assist in the installation (rather than the actual development and debugging) of applications. This utility is primarily a debugging tool.                                                                                                                                                                                                                                                            |
| 10977 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10978 | <i>prof</i>   | RCS was originally considered as part of a version control utilities portion of the scope. However, this aspect was abandoned by the standard developers. SCCS is now included as an optional part of the XSI extension.                                                                                                                                                                                                                                              |
| 10979 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10980 | RCS           | Restricted editor. This was not considered by the standard developers because it never provided the level of security restriction required.                                                                                                                                                                                                                                                                                                                           |
| 10981 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10982 | <i>red</i>    | Restricted shell. This was not considered by the standard developers because it does not provide the level of security restriction that is implied by historical documentation.                                                                                                                                                                                                                                                                                       |
| 10983 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10984 | <i>rsh</i>    | The intent of the various software development utilities was to assist in the installation (rather than the actual development and debugging) of applications. This utility is primarily a debugging tool. Furthermore, some useful aspects of <i>sdb</i> are very hardware-specific.                                                                                                                                                                                 |
| 10985 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10986 | <i>sdb</i>    | The “side-by-side <i>diff</i> ” utility from System V was omitted because it is used infrequently, and even less so by conforming applications. Despite being in System V, it is not in the SVID or XPG.                                                                                                                                                                                                                                                              |
| 10987 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10988 | <i>sdiff</i>  | Any of the numerous “shell archivers” were excluded because they did not meet the requirement of existing practice.                                                                                                                                                                                                                                                                                                                                                   |
| 10989 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10990 | <i>shar</i>   | This utility is terminal-oriented and is not useful from shell scripts or typical application programs. The job control aspects of the shell command language are generally more useful.                                                                                                                                                                                                                                                                              |
| 10991 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10992 | <i>shl</i>    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10993 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10994 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10995 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10996 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10997 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10998 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 10999 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 11000 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 11001 |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |

|       |               |                                                                                                                                                                                                                                                                                                                                 |
|-------|---------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 11002 | <i>size</i>   | The intent of the various software development utilities was to assist in the installation (rather than the actual development and debugging) of applications. This utility is primarily a debugging tool.                                                                                                                      |
| 11003 |               |                                                                                                                                                                                                                                                                                                                                 |
| 11004 |               |                                                                                                                                                                                                                                                                                                                                 |
| 11005 | <i>spell</i>  | This utility is not useful from shell scripts or typical application programs. The <i>spell</i> utility was considered, but was omitted because there is no known technology that can be used to make it recognize general language for user-specified input without providing a complete dictionary along with the input file. |
| 11006 |               |                                                                                                                                                                                                                                                                                                                                 |
| 11007 |               |                                                                                                                                                                                                                                                                                                                                 |
| 11008 |               |                                                                                                                                                                                                                                                                                                                                 |
| 11009 | <i>su</i>     | This utility is not useful from shell scripts or typical application programs. (There was also sentiment to avoid security-related utilities.)                                                                                                                                                                                  |
| 11010 |               |                                                                                                                                                                                                                                                                                                                                 |
| 11011 | <i>sum</i>    | This utility was renamed <i>cksum</i> .                                                                                                                                                                                                                                                                                         |
| 11012 | <i>tar</i>    | This has been replaced by <i>pax</i> , for reasons explained in the rationale for that utility. 1                                                                                                                                                                                                                               |
| 11013 | <i>unpack</i> | This compression program was considered inferior to <i>uncompress</i> .                                                                                                                                                                                                                                                         |
| 11014 | <i>wall</i>   | This utility is terminal-oriented and is not useful in shell scripts or typical applications. It is generally used only by system administrators.                                                                                                                                                                               |
| 11015 |               |                                                                                                                                                                                                                                                                                                                                 |





11017 / *Rationale (Informative)*

11018 **Part D:**

11019 **Portability Considerations**

11020 *The Open Group*

11021 *The Institute of Electrical and Electronics Engineers, Inc.*



## Portability Considerations (Informative)

11023

11024 This section contains information to satisfy various international requirements:

- 11025 • Section D.1 describes perceived user requirements.
- 11026 • Section D.2 (on page 274) indicates how the facilities of IEEE Std 1003.1-2001 satisfy those
- 11027 requirements.
- 11028 • Section D.3 (on page 281) offers guidance to writers of profiles on how the configurable
- 11029 options, limits, and optional behavior of IEEE Std 1003.1-2001 should be cited in profiles.

### 11030 D.1 User Requirements

11031 This section describes the user requirements that were perceived by the developers of  
 11032 IEEE Std 1003.1-2001. The primary source for these requirements was an analysis of historical  
 11033 practice in widespread use, as typified by the base documents listed in Section A.1.1 (on page 3).

11034 IEEE Std 1003.1-2001 addresses the needs of users requiring open systems solutions for source  
 11035 code portability of applications. It currently addresses users requiring open systems solutions  
 11036 for source-code portability of applications involving multi-programming and process  
 11037 management (creating processes, signaling, and so on); access to files and directories in a  
 11038 hierarchy of file systems (opening, reading, writing, deleting files, and so on); access to  
 11039 asynchronous communications ports and other special devices; access to information about  
 11040 other users of the system; facilities supporting applications requiring bounded (realtime)  
 11041 response.

11042 The following users are identified for IEEE Std 1003.1-2001:

- 11043 • Those employing applications written in high-level languages, such as C, Ada, or FORTRAN.
- 11044 • Users who desire conforming applications that do not necessarily require the characteristics
- 11045 of high-level languages (for example, the speed of execution of compiled languages or the
- 11046 relative security of source code intellectual property inherent in the compilation process).
- 11047 • Users who desire conforming applications that can be developed quickly and can be
- 11048 modified readily without the use of compilers and other system components that may be
- 11049 unavailable on small systems or those without special application development capabilities.
- 11050 • Users who interact with a system to achieve general-purpose time-sharing capabilities
- 11051 common to most business or government offices or academic environments: editing, filing,
- 11052 inter-user communications, printing, and so on.
- 11053 • Users who develop applications for POSIX-conformant systems.
- 11054 • Users who develop applications for UNIX systems.

11055 An acknowledged restriction on applicable users is that they are limited to the group of  
 11056 individuals who are familiar with the style of interaction characteristic of historically-derived  
 11057 systems based on one of the UNIX operating systems (as opposed to other historical systems  
 11058 with different models, such as MS/DOS, Macintosh, VMS, MVS, and so on). Typical users  
 11059 would include program developers, engineers, or general-purpose time-sharing users.

11060 The requirements of users of IEEE Std 1003.1-2001 can be summarized as a single goal:  
 11061 *application source portability*. The requirements of the user are stated in terms of the requirements

11062 of portability of applications. This in turn becomes a requirement for a standardized set of  
11063 syntax and semantics for operations commonly found on many operating systems.

11064 The following sections list the perceived requirements for application portability.

#### 11065 **D.1.1 Configuration Interrogation**

11066 An application must be able to determine whether and how certain optional features are  
11067 provided and to identify the system upon which it is running, so that it may appropriately adapt  
11068 to its environment.

11069 Applications must have sufficient information to adapt to varying behaviors of the system.

#### 11070 **D.1.2 Process Management**

11071 An application must be able to manage itself, either as a single process or as multiple processes.  
11072 Applications must be able to manage other processes when appropriate.

11073 Applications must be able to identify, control, create, and delete processes, and there must be  
11074 communication of information between processes and to and from the system.

11075 Applications must be able to use multiple flows of control with a process (threads) and  
11076 synchronize operations between these flows of control.

#### 11077 **D.1.3 Access to Data**

11078 Applications must be able to operate on the data stored on the system, access it, and transmit it  
11079 to other applications. Information must have protection from unauthorized or accidental access  
11080 or modification.

#### 11081 **D.1.4 Access to the Environment**

11082 Applications must be able to access the external environment to communicate their input and  
11083 results.

#### 11084 **D.1.5 Access to Determinism and Performance Enhancements**

11085 Applications must have sufficient control of resource allocation to ensure the timeliness of  
11086 interactions with external objects.

#### 11087 **D.1.6 Operating System-Dependent Profile**

11088 The capabilities of the operating system may make certain optional characteristics of the base  
11089 language in effect no longer optional, and this should be specified.

#### 11090 **D.1.7 I/O Interaction**

11091 The interaction between the C language I/O subsystem (*stdio*) and the I/O subsystem of  
11092 IEEE Std 1003.1-2001 must be specified.

**11093 D.1.8 Internationalization Interaction**

11094 The effects of the environment of IEEE Std 1003.1-2001 on the internationalization facilities of the  
11095 C language must be specified.

**11096 D.1.9 C-Language Extensions**

11097 Certain functions in the C language must be extended to support the additional capabilities  
11098 provided by IEEE Std 1003.1-2001.

**11099 D.1.10 Command Language**

11100 Users should be able to define procedures that combine simple tools and/or applications into  
11101 higher-level components that perform to the specific needs of the user. The user should be able  
11102 to store, recall, use, and modify these procedures. These procedures should employ a powerful  
11103 command language that is used for recurring tasks in conforming applications (scripts) in the  
11104 same way that it is used interactively to accomplish one-time tasks. The language and the  
11105 utilities that it uses must be consistent between systems to reduce errors and retraining.

**11106 D.1.11 Interactive Facilities**

11107 Use the system to accomplish individual tasks at an interactive terminal. The interface should be  
11108 consistent, intuitive, and offer usability enhancements to increase the productivity of terminal  
11109 users, reduce errors, and minimize retraining costs. Online documentation or usage assistance  
11110 should be available.

**11111 D.1.12 Accomplish Multiple Tasks Simultaneously**

11112 Access applications and interactive facilities from a single terminal without requiring serial  
11113 execution: switch between multiple interactive tasks; schedule one-time or periodic background  
11114 work; display the status of all work in progress or scheduled; influence the priority scheduling of  
11115 work, when authorized.

**11116 D.1.13 Complex Data Manipulation**

11117 Manipulate data in files in complex ways: sort, merge, compare, translate, edit, format, pattern  
11118 match, select subsets (strings, columns, fields, rows, and so on). These facilities should be  
11119 available to both conforming applications and interactive users.

**11120 D.1.14 File Hierarchy Manipulation**

11121 Create, delete, move/rename, copy, backup/archive, and display files and directories. These  
11122 facilities should be available to both conforming applications and interactive users.

**11123 D.1.15 Locale Configuration**

11124 Customize applications and interactive sessions for the cultural and language conventions of the  
11125 user. Employ a wide variety of standard character encodings. These facilities should be available  
11126 to both conforming applications and interactive users.

**11127 D.1.16 Inter-User Communication**

11128 Send messages or transfer files to other users on the same system or other systems on a network.  
 11129 These facilities should be available to both conforming applications and interactive users.

**11130 D.1.17 System Environment**

11131 Display information about the status of the system (activities of users and their interactive and  
 11132 background work, file system utilization, system time, configuration, and presence of optional  
 11133 facilities) and the environment of the user (terminal characteristics, and so on). Inform the  
 11134 system operator/administrator of problems. Control access to user files and other resources.

**11135 D.1.18 Printing**

11136 Output files on a variety of output device classes, accessing devices on local or network-  
 11137 connected systems. Control (or influence) the formatting, priority scheduling, and output  
 11138 distribution of work. These facilities should be available to both conforming applications and  
 11139 interactive users.

**11140 D.1.19 Software Development**

11141 Develop (create and manage source files, compile/interpret, debug) portable open systems  
 11142 applications and package them for distribution to, and updating of, other systems.

**11143 D.2 Portability Capabilities**

11144 This section describes the significant portability capabilities of IEEE Std 1003.1-2001 and  
 11145 indicates how the user requirements listed in Section D.1 (on page 271) are addressed. The  
 11146 capabilities are listed in the same format as the preceding user requirements; they are  
 11147 summarized below:

- 11148 • Configuration Interrogation
- 11149 • Process Management
- 11150 • Access to Data
- 11151 • Access to the Environment
- 11152 • Access to Determinism and Performance Enhancements
- 11153 • Operating System-Dependent Profile
- 11154 • I/O Interaction
- 11155 • Internationalization Interaction
- 11156 • C-Language Extensions
- 11157 • Command Language
- 11158 • Interactive Facilities
- 11159 • Accomplish Multiple Tasks Simultaneously
- 11160 • Complex Data Manipulation
- 11161 • File Hierarchy Manipulation

- 11162 • Locale Configuration
- 11163 • Inter-User Communication
- 11164 • System Environment
- 11165 • Printing
- 11166 • Software Development

### 11167 D.2.1 Configuration Interrogation

11168 The *uname()* operation provides basic identification of the system. The *sysconf()*, *pathconf()*, and  
 11169 *fpathconf()* functions and the *getconf* utility provide means to interrogate the implementation to  
 11170 determine how to adapt to the environment in which it is running. These values can be either  
 11171 static (indicating that all instances of the implementation have the same value) or dynamic  
 11172 (indicating that different instances of the implementation have the different values, or that the  
 11173 value may vary for other reasons, such as reconfiguration).

### 11174 Unsatisfied Requirements

11175 None directly. However, as new areas are added, there will be a need for additional capability in  
 11176 this area.

### 11177 D.2.2 Process Management

11178 The *fork()*, *exec* family, *posix\_spawn()*, and *posix\_spawnnp()* functions provide for the creation of  
 11179 new processes or the insertion of new applications into existing processes. The *\_Exit()*, *\_exit()*,  
 11180 *exit()*, and *abort()* functions allow for the termination of a process by itself. The *wait()* and  
 11181 *waitpid()* functions allow one process to deal with the termination of another.

11182 The *times()* function allows for basic measurement of times used by a process. Various  
 11183 functions, including *fstat()*, *getegid()*, *geteuid()*, *getgid()*, *getgrgid()*, *getgrnam()*, *getlogin()*,  
 11184 *getpid()*, *getppid()*, *getpwnam()*, *getpwuid()*, *getuid()*, *lstat()*, and *stat()*, provide for access to the  
 11185 identifiers of processes and the identifiers and names of owners of processes (and files).

11186 The various functions operating on environment variables provide for communication of  
 11187 information (primarily user-configurable defaults) from a parent to child processes.

11188 The operations on the current working directory control and interrogate the directory from  
 11189 which relative filename searches start. The *umask()* function controls the default protections  
 11190 applied to files created by the process.

11191 The *alarm()*, *pause()*, *sleep()*, *ualarm()*, and *usleep()* operations allow the process to suspend until  
 11192 a timer has expired or to be notified when a period of time has elapsed. The *time()* operation  
 11193 interrogates the current time and date.

11194 The signal mechanism provides for communication of events either from other processes or  
 11195 from the environment to the application, and the means for the application to control the effect  
 11196 of these events. The mechanism provides for external termination of a process and for a process  
 11197 to suspend until an event occurs. The mechanism also provides for a value to be associated with  
 11198 an event.

11199 Job control provides a means to group processes and control them as groups, and to control their  
 11200 access to the function between the user and the system (the “controlling terminal”). It also  
 11201 provides the means to suspend and resume processes.

11202 The Process Scheduling option provides control of the scheduling and priority of a process.

11203 The Message Passing option provides a means for interprocess communication involving small  
11204 amounts of data.

11205 The Memory Management facilities provide control of memory resources and for the sharing of  
11206 memory. This functionality is mandatory on XSI-conformant systems.

11207 The Threads facilities provide multiple flows of control with a process (threads),  
11208 synchronization between threads, association of data with threads, and controlled cancellation  
11209 of threads.

11210 The XSI interprocess communications functionality provide an alternate set of facilities to  
11211 manipulate semaphores, message queues, and shared memory. These are provided on XSI-  
11212 conformant systems to support conforming applications developed to run on UNIX systems.

### 11213 D.2.3 Access to Data

11214 The *open()*, *close()*, *fclose()*, *fopen()*, and *pipe()* functions provide for access to files and data.  
11215 Such files may be regular files, interprocess data channels (pipes), or devices. Additional types  
11216 of objects in the file system are permitted and are being contemplated for standardization.

11217 The *access()*, *chmod()*, *chown()*, *dup()*, *dup2()*, *fchmod()*, *fcntl()*, *fstat()*, *ftruncate()*, *lstat()*,  
11218 *readlink()*, *realpath()*, *stat()*, and *utime()* functions allow for control and interrogation of file and  
11219 file-related objects (including symbolic links), and their ownership, protections, and timestamps.

11220 The *fgetc()*, *fputc()*, *fread()*, *fseek()*, *fsetpos()*, *fwrite()*, *getc()*, *getchar()*, *lseek()*, *putchar()*, *putc()*,  
11221 *read()*, and *write()* functions provide for data transfer from the application to files (in all their  
11222 forms).

11223 The *closedir()*, *link()*, *mkdir()*, *opendir()*, *readdir()*, *rename()*, *rmdir()*, *rewinddir()*, and *unlink()*  
11224 functions provide for a complete set of operations on directories. Directories can arbitrarily  
11225 contain other directories, and a single file can be mentioned in more than one directory.

11226 The file-locking mechanism provides for advisory locking (protection during transactions) of  
11227 ranges of bytes (in effect, records) in a file.

11228 The *confstr()*, *fpathconf()*, *pathconf()*, and *sysconf()* functions provide for enquiry as to the  
11229 behavior of the system where variability is permitted.

11230 The Synchronized Input and Output option provides for assured commitment of data to media.

11231 The Asynchronous Input and Output option provides for initiation and control of asynchronous  
11232 data transfers.

### 11233 D.2.4 Access to the Environment

11234 The operations and types in the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11,  
11235 General Terminal Interface are provided for access to asynchronous serial devices. The primary  
11236 intended use for these is the controlling terminal for the application (the interaction point  
11237 between the user and the system). They are general enough to be used to control any  
11238 asynchronous serial device. The functions are also general enough to be used with many other  
11239 device types as a user interface when some emulation is provided.

11240 Less detailed access is provided for other device types, but in many instances an application  
11241 need not know whether an object in the file system is a device or a regular file to operate  
11242 correctly.



11243 **Unsatisfied Requirements**

11244 Detailed control of common device classes, specifically magnetic tape, is not provided.

11245 **D.2.5 Bounded (Realtime) Response**

11246 The Realtime Signals Extension provides queued signals and the prioritization of the handling of  
 11247 signals. The SCHED\_FIFO, SCHED\_SPORADIC, and SCHED\_RR scheduling policies provide  
 11248 control over processor allocation. The Semaphores option provides high-performance  
 11249 synchronization. The Memory Management functions provide memory locking for control of  
 11250 memory allocation, file mapping for high-performance, and shared memory for high-  
 11251 performance interprocess communication. The Message Passing option provides for interprocess  
 11252 communication without being dependent on shared memory.

11253 The Timers option provides a high resolution function called *nanosleep()* with a finer resolution  
 11254 than the *sleep()* function.

11255 The Typed Memory Objects option, the Monotonic Clock option, and the Timeouts option  
 11256 provide further facilities for applications to use to obtain predictable bounded response.

11257 **D.2.6 Operating System-Dependent Profile**

11258 IEEE Std 1003.1-2001 makes no distinction between text and binary files. The values of  
 11259 EXIT\_SUCCESS and EXIT\_FAILURE are further defined.

11260 **Unsatisfied Requirements**

11261 None known, but the ISO C standard may contain some additional options that could be  
 11262 specified.

11263 **D.2.7 I/O Interaction**

11264 IEEE Std 1003.1-2001 defines how each of the ISO C standard *stdio* functions interact with the  
 11265 POSIX.1 operations, typically specifying the behavior in terms of POSIX.1 operations.

11266 **Unsatisfied Requirements**

11267 None.

11268 **D.2.8 Internationalization Interaction**

11269 The IEEE Std 1003.1-2001 environment operations provide a means to define the environment  
 11270 for *setlocale()* and time functions such as *ctime()*. The *tzset()* function is provided to set time  
 11271 conversion information.

11272 The *nl\_langinfo()* function is provided as an XSI extension to query locale-specific cultural  
 11273 settings.

11274 **Unsatisfied Requirements**

11275 None.

**11276 D.2.9 C-Language Extensions**

11277 The *setjmp()* and *longjmp()* functions are not defined to be cognizant of the signal masks defined  
 11278 for POSIX.1. The *sigsetjmp()* and *siglongjmp()* functions are provided to fill this gap.

11279 The *\_setjmp()* and *\_longjmp()* functions are provided as XSI extensions to support historic  
 11280 practice.

**11281 Unsatisfied Requirements**

11282 None.

**11283 D.2.10 Command Language**

11284 The shell command language, as described in the Shell and Utilities volume of  
 11285 IEEE Std 1003.1-2001, Chapter 2, Shell Command Language, is a common language useful in  
 11286 batch scripts, through an API to high-level languages (for the C-Language Binding option,  
 11287 *system()* and *popen()*) and through an interactive terminal (see the *sh* utility). The shell language  
 11288 has many of the characteristics of a high-level language, but it has been designed to be more  
 11289 suitable for user terminal entry and includes interactive debugging facilities. Through the use of  
 11290 pipelining, many complex commands can be constructed from combinations of data filters and  
 11291 other common components. Shell scripts can be created, stored, recalled, and modified by the  
 11292 user with simple editors.

11293 In addition to the basic shell language, the following utilities offer features that simplify and  
 11294 enhance programmatic access to the utilities and provide features normally found only in high-  
 11295 level languages: *basename*, *bc*, *command*, *dirname*, *echo*, *env*, *expr*, *false*, *printf*, *read*, *sleep*, *tee*, *test*,  
 11296 *time\**,<sup>2</sup> *true*, *wait*, *xargs*, and all of the special built-in utilities in the Shell and Utilities volume of  
 11297 IEEE Std 1003.1-2001, Section 2.14, Special Built-In Utilities.

**11298 Unsatisfied Requirements**

11299 None.

**11300 D.2.11 Interactive Facilities**

11301 The utilities offer a common style of command-line interface through conformance to the Utility  
 11302 Syntax Guidelines (see the Base Definitions volume of IEEE Std 1003.1-2001, Section 12.2, Utility  
 11303 Syntax Guidelines) and the common utility defaults (see the Shell and Utilities volume of  
 11304 IEEE Std 1003.1-2001, Section 1.11, Utility Description Defaults). The *sh* utility offers an  
 11305 interactive command-line history and editing facility. The following utilities in the User  
 11306 Portability Utilities option have been customized for interactive use: *alias*, *ex*, *fc*, *mailx*, *more*, *talk*,  
 11307 *vi*, *unalias*, and *write*; the *man* utility offers online access to system documentation.

11308 \_\_\_\_\_

11309 2. The utilities listed with an asterisk here and later in this section are present only on systems which support the User Portability  
 11310 Utilities option. There may be further restrictions on the utilities offered with various configuration option combinations; see the  
 11311 individual utility descriptions.

11312 **Unsatisfied Requirements**

11313 The command line interface to individual utilities is as intuitive and consistent as historical  
 11314 practice allows. Work underway based on graphical user interfaces may be more suitable for  
 11315 novice or occasional users of the system.

11316 **D.2.12 Accomplish Multiple Tasks Simultaneously**

11317 The shell command language offers background processing through the asynchronous list  
 11318 command form; see the Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.9, Shell  
 11319 Commands. The *nohup* utility makes background processing more robust and usable. The *kill*  
 11320 utility can terminate background jobs. When the User Portability Utilities option is supported,  
 11321 the following utilities allow manipulation of jobs: *bg*, *fg*, and *jobs*. Also, if the User Portability  
 11322 Utilities option is supported, the following can support periodic job scheduling, control, and  
 11323 display: *at*, *batch*, *crontab*, *nice*, *ps*, and *renice*.

11324 **Unsatisfied Requirements**

11325 Terminals with multiple windows may be more suitable for some multi-tasking interactive uses  
 11326 than the job control approach in IEEE Std 1003.1-2001. See the comments on graphical user  
 11327 interfaces in Section D.2.11 (on page 278). The *nice* and *renice* utilities do not necessarily take  
 11328 advantage of complex system scheduling algorithms that are supported by the realtime options  
 11329 within IEEE Std 1003.1-2001.

11330 **D.2.13 Complex Data Manipulation**

11331 The following utilities address user requirements in this area: *asa*, *awk*, *bc*, *cmp*, *comm*, *csplit\**, *cut*,  
 11332 *dd*, *diff*, *ed*, *ex\**, *expand\**, *expr*, *find*, *fold*, *grep*, *head*, *join*, *od*, *paste*, *pr*, *printf*, *sed*, *sort*, *split\**, *tabs\**, *tail*,  
 11333 *tr*, *unexpand\**, *uniq*, *uudecode\**, *uuencode\**, and *wc*.

11334 **Unsatisfied Requirements**

11335 Sophisticated text formatting utilities, such as *troff* or *TeX*, are not included. Standards work in  
 11336 the area of SGML may satisfy this.

11337 **D.2.14 File Hierarchy Manipulation**

11338 The following utilities address user requirements in this area: *basename*, *cd*, *chgrp*, *chmod*, *chown*,  
 11339 *cksum*, *cp*, *dd*, *df\**, *diff*, *dirname*, *du\**, *find*, *ls*, *ln*, *mkdir*, *mkfifo*, *mv*, *patch\**, *pathchk*, *pax*, *pwd*, *rm*, *rmdir*,  
 11340 *test*, and *touch*.

11341 **Unsatisfied Requirements**

11342 Some graphical user interfaces offer more intuitive file manager components that allow file  
 11343 manipulation through the use of icons for novice users.

**11344 D.2.15 Locale Configuration**

11345 The standard utilities are affected by the various *LC\_* variables to achieve locale-dependent  
 11346 operation: character classification, collation sequences, regular expressions and shell pattern  
 11347 matching, date and time formats, numeric formatting, and monetary formatting. When the  
 11348 POSIX2\_LOCALEDEF option is supported, applications can provide their own locale definition  
 11349 files. The following utilities address user requirements in this area: *date*, *ed*, *ex\**, *find*, *grep*, *locale*,  
 11350 *localedef*, *more\**, *sed*, *sh*, *sort*, *tr*, *uniq*, and *vi\**.

11351 The *iconv()*, *iconv\_close()*, and *iconv\_open()* functions are available to allow an application to  
 11352 convert character data between supported character sets.

11353 The *gencat* utility and the *catopen()*, *catclose()*, and *catgets()* functions for message catalog  
 11354 manipulation are available on XSI-conformant systems.

**11355 Unsatisfied Requirements**

11356 Some aspects of multi-byte character and state-encoded character encodings have not yet been  
 11357 addressed. The C-language functions, such as *getopt()*, are generally limited to single-byte  
 11358 characters. The effect of the *LC\_MESSAGES* variable on message formats is only suggested at  
 11359 this time.

**11360 D.2.16 Inter-User Communication**

11361 The following utilities address user requirements in this area: *cksum*, *mailx\**, *mesg\**, *patch\**, *pax*,  
 11362 *talk\**, *uudecode\**, *uuencode\**, *who\**, and *write\**.

11363 The historical UUCP utilities are included on XSI-conformant systems.

**11364 Unsatisfied Requirements**

11365 None.

**11366 D.2.17 System Environment**

11367 The following utilities address user requirements in this area: *chgrp*, *chmod*, *chown*, *df\**, *du\**, *env*,  
 11368 *getconf*, *id*, *logger*, *logname*, *mesg\**, *newgrp\**, *ps\**, *stty*, *tput\**, *tty*, *umask*, *uname*, and *who\**.

11369 The *closelog()*, *openlog()*, *setlogmask()*, and *syslog()* functions provide System Logging facilities  
 11370 on XSI-conformant systems; these are analogous to the *logger* utility.

**11371 Unsatisfied Requirements**

11372 None.

**11373 D.2.18 Printing**

11374 The following utilities address user requirements in this area: *pr* and *lp*.

**11375 Unsatisfied Requirements**

11376 There are no features to control the formatting or scheduling of the print jobs.

**11377 D.2.19 Software Development**

11378 The following utilities address user requirements in this area: *ar*, *asa*, *awk*, *c99*, *ctags\**, *fort77*,  
 11379 *getconf*, *getopts*, *lex*, *localedef*, *make*, *nm\**, *od*, *patch\**, *pax*, *strings\**, *strip*, *time\**, and *yacc*.

11380 The *system()*, *popen()*, *pclose()*, *regcomp()*, *regexexec()*, *regerror()*, *regfree()*, *fnmatch()*, *getopt()*,  
 11381 *glob()*, *globfree()*, *wordexp()*, and *wordfree()* functions allow C-language programmers to access  
 11382 some of the interfaces used by the utilities, such as argument processing, regular expressions,  
 11383 and pattern matching.

11384 The SCCS source-code control system utilities are available on systems supporting the XSI  
 11385 Development option.

**11386 Unsatisfied Requirements**

11387 There are no language-specific development tools related to languages other than C and  
 11388 FORTRAN. The C tools are more complete and varied than the FORTRAN tools. There is no  
 11389 data dictionary or other CASE-like development tools.

**11390 D.2.20 Future Growth**

11391 It is arguable whether or not all functionality to support applications is potentially within the  
 11392 scope of IEEE Std 1003.1-2001. As a simple matter of practicality, it cannot be. Areas such as  
 11393 graphics, application domain-specific functionality, windowing, and so on, should be in unique  
 11394 standards. As such, they are properly “Unsatisfied Requirements” in terms of providing fully  
 11395 conforming applications, but ones which are outside the scope of IEEE Std 1003.1-2001.

11396 However, as the standards evolve, certain functionality once considered “exotic” enough to be  
 11397 part of a separate standard become common enough to be included in a core standard such as  
 11398 this. Realtime and networking, for example, have both moved from separate standards (with  
 11399 much difficult cross-referencing) into IEEE Std 1003.1 over time, and although no specific areas  
 11400 have been identified for inclusion in future revisions, such inclusions seem likely.

**11401 D.3 Profiling Considerations**

11402 This section offers guidance to writers of profiles on how the configurable options, limits, and  
 11403 optional behavior of IEEE Std 1003.1-2001 should be cited in profiles. Profile writers should  
 11404 consult the general guidance in POSIX.0 when writing POSIX Standardized Profiles.

11405 The information in this section is an inclusive list of features that should be considered by profile  
 11406 writers. Subsetting of IEEE Std 1003.1-2001 should follow the Base Definitions volume of  
 11407 IEEE Std 1003.1-2001, Section 2.1.5.1, Subprofiling Considerations. A set of profiling options is  
 11408 described in Appendix E (on page 295).

**11409 D.3.1 Configuration Options**

11410 There are two set of options suggested by IEEE Std 1003.1-2001: those for POSIX-conforming  
 11411 systems and those for X/Open System Interface (XSI) conformance. The requirements for XSI  
 11412 conformance are documented in the Base Definitions volume of IEEE Std 1003.1-2001 and not  
 11413 discussed further here, as they superset the POSIX conformance requirements.

### 11414 D.3.2 Configuration Options (Shell and Utilities)

11415 There are three broad optional configurations for the Shell and Utilities volume of  
 11416 IEEE Std 1003.1-2001: basic execution system, development system, and user portability  
 11417 interactive system. The options to support these, and other minor configuration options, are  
 11418 listed in the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 2, Conformance. Profile  
 11419 writers should consult the following list and the comments concerning user requirements  
 11420 addressed by various components in Section D.2 (on page 274).

#### 11421 POSIX2\_UPE

11422 The system supports the User Portability Utilities option.

11423 This option is a requirement for a user portability interactive system. It is required  
 11424 frequently except for those systems, such as embedded realtime or dedicated application  
 11425 systems, that support little or no interactive time-sharing work by users or operators. XSI-  
 11426 conformant systems support this option.

#### 11427 POSIX2\_SW\_DEV

11428 The system supports the Software Development Utilities option.

11429 This option is required by many systems, even those in which actual software development  
 11430 does not occur. The *make* utility, in particular, is required by many application software  
 11431 packages as they are installed onto the system. If POSIX2\_C\_DEV is supported,  
 11432 POSIX2\_SW\_DEV is almost a mandatory requirement because of *ar* and *make*.

#### 11433 POSIX2\_C\_BIND

11434 The system supports the C-Language Bindings option.

11435 This option is required on some implementations developing complex C applications or on  
 11436 any system installing C applications in source form that require the functions in this option.  
 11437 The *system()* and *popen()* functions, in particular, are widely used by applications; the  
 11438 others are rather more specialized.

#### 11439 POSIX2\_C\_DEV

11440 The system supports the C-Language Development Utilities option.

11441 This option is required by many systems, even those in which actual C-language software  
 11442 development does not occur. The *c99* utility, in particular, is required by many application  
 11443 software packages as they are installed onto the system. The *lex* and *yacc* utilities are used  
 11444 less frequently.

#### 11445 POSIX2\_FORT\_DEV

11446 The system supports the FORTRAN Development Utilities option

11447 As with C, this option is needed on any system developing or installing FORTRAN  
 11448 applications in source form.

#### 11449 POSIX2\_FORT\_RUN

11450 The system supports the FORTRAN Runtime Utilities option.

11451 This option is required for some FORTRAN applications that need the *asa* utility to convert  
 11452 Hollerith printing statement output. It is unknown how frequently this occurs.

#### 11453 POSIX2\_LOCALEDEF

11454 The system supports the creation of locales.

11455 This option is needed if applications require their own customized locale definitions to  
 11456 operate. It is presently unknown whether many applications are dependent on this.  
 11457 However, the option is virtually mandatory for systems in which internationalized  
 11458 applications are developed.

|       |                                                                                             |   |
|-------|---------------------------------------------------------------------------------------------|---|
| 11459 | XSI-conformant systems support this option.                                                 |   |
| 11460 | POSIX2_PBS                                                                                  |   |
| 11461 | The system supports the Batch Environment Services and Utilities option.                    | 1 |
| 11462 | POSIX2_PBS_ACCOUNTING                                                                       |   |
| 11463 | The system supports the optional feature of accounting within the Batch Environment         | 1 |
| 11464 | Services and Utilities option. It will be required in servers that implement the optional   | 1 |
| 11465 | feature of accounting.                                                                      | 1 |
| 11466 | POSIX2_PBS_CHECKPOINT                                                                       |   |
| 11467 | The system supports the optional feature of checkpoint/restart within the Batch             | 1 |
| 11468 | Environment Services and Utilities option.                                                  | 1 |
| 11469 | POSIX2_PBS_LOCATE                                                                           |   |
| 11470 | The system supports the optional feature of locating batch jobs within the Batch            | 1 |
| 11471 | Environment Services and Utilities option.                                                  | 1 |
| 11472 | POSIX2_PBS_MESSAGE                                                                          |   |
| 11473 | The system supports the optional feature of sending messages to batch jobs within the       | 1 |
| 11474 | Batch Environment Services and Utilities option.                                            | 1 |
| 11475 | POSIX2_PBS_TRACK                                                                            |   |
| 11476 | The system supports the optional feature of tracking batch jobs within the Batch            | 1 |
| 11477 | Environment Services and Utilities option.                                                  | 1 |
| 11478 | POSIX2_CHAR_TERM                                                                            |   |
| 11479 | The system supports at least one terminal type capable of all operations described in       |   |
| 11480 | IEEE Std 1003.1-2001.                                                                       |   |
| 11481 | On systems with POSIX2_UPE, this option is almost always required. It was developed         |   |
| 11482 | solely to allow certain specialized vendors and user applications to bypass the requirement |   |
| 11483 | for general-purpose asynchronous terminal support. For example, an application and          |   |
| 11484 | system that was suitable for block-mode terminals, such as IBM 3270s, would not need this   |   |
| 11485 | option.                                                                                     |   |
| 11486 | XSI-conformant systems support this option.                                                 |   |

### 11487 D.3.3 Configurable Limits

|       |                                                                                                        |  |
|-------|--------------------------------------------------------------------------------------------------------|--|
| 11488 | Very few of the limits need to be increased for profiles. No profile can cite lower values.            |  |
| 11489 | {POSIX2_BC_BASE_MAX}                                                                                   |  |
| 11490 | {POSIX2_BC_DIM_MAX}                                                                                    |  |
| 11491 | {POSIX2_BC_SCALE_MAX}                                                                                  |  |
| 11492 | {POSIX2_BC_STRING_MAX}                                                                                 |  |
| 11493 | No increase is anticipated for any of these <i>bc</i> values, except for very specialized applications |  |
| 11494 | involving huge numbers.                                                                                |  |
| 11495 | {POSIX2_COLL_WEIGHTS_MAX}                                                                              |  |
| 11496 | Some natural languages with complex collation requirements require an increase from the                |  |
| 11497 | default 2 to 4; no higher numbers are anticipated.                                                     |  |
| 11498 | {POSIX2_EXPR_NEST_MAX}                                                                                 |  |
| 11499 | No increase is anticipated.                                                                            |  |
| 11500 | {POSIX2_LINE_MAX}                                                                                      |  |
| 11501 | This number is much larger than most historical applications have been able to use. At some            |  |
| 11502 | future time, applications may be rewritten to take advantage of even larger values.                    |  |

11503 {POSIX2\_RE\_DUP\_MAX}  
 11504 No increase is anticipated.

11505 {POSIX2\_VERSION}  
 11506 This is actually not a limit, but a standard version stamp. Generally, a profile should specify  
 11507 the Shell and Utilities volume of IEEE Std 1003.1-2001, Chapter 2, Shell Command Language  
 11508 by name in the normative references section, not this value.

#### 11509 **D.3.4 Configuration Options (System Interfaces)**

11510 {NGROUPS\_MAX}  
 11511 A non-zero value indicates that the implementation supports supplementary groups.

11512 This option is needed where there is a large amount of shared use of files, but where a  
 11513 certain amount of protection is needed. Many profiles<sup>3</sup> are known to require this option; it  
 11514 should only be required if needed, but it should never be prohibited.

11515 \_POSIX\_ADVISORY\_INFO  
 11516 The system provides advisory information for file management.

11517 This option allows the application to specify advisory information that can be used to  
 11518 achieve better or even deterministic response time in file manager or input and output  
 11519 operations.

11520 \_POSIX\_ASYNCHRONOUS\_IO  
 11521 The system provides concurrent process execution and input and output transfers.

11522 This option was created to support historical systems that did not provide the feature. It  
 11523 should only be required if needed, but it should never be prohibited.

11524 \_POSIX\_BARRIERS  
 11525 The system supports barrier synchronization.

11526 This option was created to allow efficient synchronization of multiple parallel threads in  
 11527 multi-processor systems in which the operation is supported in part by the hardware  
 11528 architecture.

11529 \_POSIX\_CHOWN\_RESTRICTED  
 11530 The system restricts the right to “give away” files to other users.

11531 This option should be carefully investigated before it is required. Some applications expect  
 11532 that they can change the ownership of files in this way. It is provided where either security  
 11533 or system account requirements cause this ability to be a problem. It is also known to be  
 11534 specified in many profiles.

11535 \_POSIX\_CLOCK\_SELECTION  
 11536 The system supports the Clock Selection option.

11537 This option allows applications to request a high resolution sleep in order to suspend a  
 11538 thread during a relative time interval, or until an absolute time value, using the desired  
 11539 clock. It also allows the application to select the clock used in a *pthread\_cond\_timedwait()*  
 11540 function call.

11541 \_\_\_\_\_

11542 3. There are no formally approved profiles of IEEE Std 1003.1-2001 at the time of publication; the reference here is to various  
 11543 profiles generated by private bodies or governments.



11544 `_POSIX_CPUTIME`

11545 The system supports the Process CPU-Time Clocks option.

11546 This option allows applications to use a new clock that measures the execution times of  
 11547 processes or threads, and the possibility to create timers based upon these clocks, for  
 11548 runtime detection (and treatment) of execution time overruns.

11549 `_POSIX_FSYNC`

11550 The system supports file synchronization requests.

11551 This option was created to support historical systems that did not provide the feature.  
 11552 Applications that are expecting guaranteed completion of their input and output operations  
 11553 should require the `_POSIX_SYNC_IO` option. This option should never be prohibited.

11554 XSI-conformant systems support this option.

11555 `_POSIX_IPV6`

11556 The system supports facilities related to Internet Protocol Version 6 (IPv6).

11557 This option was created to allow systems to transition to IPv6.

11558 `_POSIX_JOB_CONTROL`

11559 Job control facilities are mandatory in IEEE Std 1003.1-2001.

11560 The option was created primarily to support historical systems that did not provide the  
 11561 feature. Many existing profiles now require it; it should only be required if needed, but it  
 11562 should never be prohibited. Most applications that use it can run when it is not present,  
 11563 although with a degraded level of user convenience.

11564 `_POSIX_MAPPED_FILES`

11565 The system supports the mapping of regular files into the process address space.

11566 XSI-conformant systems support this option.

11567 Both this option and the Shared Memory Objects option provide shared access to memory  
 11568 objects in the process address space. The functions defined under this option provide the  
 11569 functionality of existing practice for mapping regular files. This functionality was deemed  
 11570 unnecessary, if not inappropriate, for embedded systems applications and, hence, is  
 11571 provided under this option. It should only be required if needed, but it should never be  
 11572 prohibited.

11573 `_POSIX_MEMLOCK`

11574 The system supports the locking of the address space.

11575 This option was created to support historical systems that did not provide the feature. It  
 11576 should only be required if needed, but it should never be prohibited.

11577 `_POSIX_MEMLOCK_RANGE`

11578 The system supports the locking of specific ranges of the address space.

11579 For applications that have well-defined sections that need to be locked and others that do  
 11580 not, IEEE Std 1003.1-2001 supports an optional set of functions to lock or unlock a range of  
 11581 process addresses. The following are two reasons for having a means to lock down a  
 11582 specific range:

- 11583 1. An asynchronous event handler function that must respond to external events in a  
 11584 deterministic manner such that page faults cannot be tolerated
- 11585 2. An input/output “buffer” area that is the target for direct-to-process I/O, and the  
 11586 overhead of implicit locking and unlocking for each I/O call cannot be tolerated

- 11587 It should only be required if needed, but it should never be prohibited.
- 11588 \_POSIX\_MEMORY\_PROTECTION
- 11589 The system supports memory protection.
- 11590 XSI-conformant systems support this option.
- 11591 The provision of this option typically imposes additional hardware requirements. It should
- 11592 never be prohibited.
- 11593 \_POSIX\_PRIORITIZED\_IO
- 11594 The system provides prioritization for input and output operations.
- 11595 The use of this option may interfere with the ability of the system to optimize input and
- 11596 output throughput. It should only be required if needed, but it should never be prohibited.
- 11597 \_POSIX\_MESSAGE\_PASSING
- 11598 The system supports the passing of messages between processes.
- 11599 This option was created to support historical systems that did not provide the feature. The
- 11600 functionality adds a high-performance XSI interprocess communication facility for local
- 11601 communication. It should only be required if needed, but it should never be prohibited.
- 11602 \_POSIX\_MONOTONIC\_CLOCK
- 11603 The system supports the Monotonic Clock option.
- 11604 This option allows realtime applications to rely on a monotonically increasing clock that
- 11605 does not jump backwards, and whose value does not change except for the regular ticking
- 11606 of the clock.
- 11607 \_POSIX\_PRIORITY\_SCHEDULING
- 11608 The system provides priority-based process scheduling.
- 11609 Support of this option provides predictable scheduling behavior, allowing applications to
- 11610 determine the order in which processes that are ready to run are granted access to a
- 11611 processor. It should only be required if needed, but it should never be prohibited.
- 11612 \_POSIX\_REALTIME\_SIGNALS
- 11613 The system provides prioritized, queued signals with associated data values.
- 11614 This option was created to support historical systems that did not provide the features. It
- 11615 should only be required if needed, but it should never be prohibited.
- 11616 \_POSIX\_REGEX
- 11617 Support for regular expression facilities is mandatory in IEEE Std 1003.1-2001.
- 11618 \_POSIX\_SAVED\_IDS
- 11619 Support for this feature is mandatory in IEEE Std 1003.1-2001.
- 11620 Certain classes of applications rely on it for proper operation, and there is no alternative
- 11621 short of giving the application root privileges on most implementations that did not provide
- 11622 \_POSIX\_SAVED\_IDS.
- 11623 \_POSIX\_SEMAPHORES
- 11624 The system provides counting semaphores.
- 11625 This option was created to support historical systems that did not provide the feature. It
- 11626 should only be required if needed, but it should never be prohibited.
- 11627 \_POSIX\_SHARED\_MEMORY\_OBJECTS
- 11628 The system supports the mapping of shared memory objects into the process address space.

- 11629 Both this option and the Memory Mapped Files option provide shared access to memory  
 11630 objects in the process address space. The functions defined under this option provide the  
 11631 functionality of existing practice for shared memory objects. This functionality was deemed  
 11632 appropriate for embedded systems applications and, hence, is provided under this option. It  
 11633 should only be required if needed, but it should never be prohibited.
- 11634 \_POSIX\_SHELL  
 11635 Support for the *sh* utility command line interpreter is mandatory in IEEE Std 1003.1-2001.
- 11636 \_POSIX\_SPAWN  
 11637 The system supports the spawn option.
- 11638 This option provides applications with an efficient mechanism to spawn execution of a new  
 11639 process.
- 11640 \_POSIX\_SPINLOCKS  
 11641 The system supports spin locks.
- 11642 This option was created to support a simple and efficient synchronization mechanism for  
 11643 threads executing in multi-processor systems.
- 11644 \_POSIX\_SPORADIC\_SERVER  
 11645 The system supports the sporadic server scheduling policy.
- 11646 This option provides applications with a new scheduling policy for scheduling aperiodic  
 11647 processes or threads in hard realtime applications.
- 11648 \_POSIX\_SYNCHRONIZED\_IO  
 11649 The system supports guaranteed file synchronization.
- 11650 This option was created to support historical systems that did not provide the feature.  
 11651 Applications that are expecting guaranteed completion of their input and output operations  
 11652 should require this option, rather than the File Synchronization option. It should only be  
 11653 required if needed, but it should never be prohibited.
- 11654 \_POSIX\_THREADS  
 11655 The system supports multiple threads of control within a single process.
- 11656 This option was created to support historical systems that did not provide the feature.  
 11657 Applications written assuming a multi-threaded environment would be expected to require  
 11658 this option. It should only be required if needed, but it should never be prohibited.
- 11659 XSI-conformant systems support this option.
- 11660 \_POSIX\_THREAD\_ATTR\_STACKADDR  
 11661 The system supports specification of the stack address for a created thread.
- 11662 Applications may take advantage of support of this option for performance benefits, but  
 11663 dependence on this feature should be minimized. This option should never be prohibited.
- 11664 XSI-conformant systems support this option.
- 11665 \_POSIX\_THREAD\_ATTR\_STACKSIZE  
 11666 The system supports specification of the stack size for a created thread.
- 11667 Applications may require this option in order to ensure proper execution, but such usage  
 11668 limits portability and dependence on this feature should be minimized. It should only be  
 11669 required if needed, but it should never be prohibited.
- 11670 XSI-conformant systems support this option.

- 11671     \_POSIX\_THREAD\_PRIORITY\_SCHEDULING
- 11672         The system provides priority-based thread scheduling.
- 11673         Support of this option provides predictable scheduling behavior, allowing applications to
- 11674         determine the order in which threads that are ready to run are granted access to a processor.
- 11675         It should only be required if needed, but it should never be prohibited.
- 11676     \_POSIX\_THREAD\_PRIO\_INHERIT
- 11677         The system provides mutual-exclusion operations with priority inheritance.
- 11678         Support of this option provides predictable scheduling behavior, allowing applications to
- 11679         determine the order in which threads that are ready to run are granted access to a processor.
- 11680         It should only be required if needed, but it should never be prohibited.
- 11681     \_POSIX\_THREAD\_PRIO\_PROTECT
- 11682         The system supports a priority ceiling emulation protocol for mutual-exclusion operations.
- 11683         Support of this option provides predictable scheduling behavior, allowing applications to
- 11684         determine the order in which threads that are ready to run are granted access to a processor.
- 11685         It should only be required if needed, but it should never be prohibited.
- 11686     \_POSIX\_THREAD\_PROCESS\_SHARED
- 11687         The system provides shared access among multiple processes to synchronization objects.
- 11688         This option was created to support historical systems that did not provide the feature. It
- 11689         should only be required if needed, but it should never be prohibited.
- 11690         XSI-conformant systems support this option.
- 11691     \_POSIX\_THREAD\_SAFE\_FUNCTIONS
- 11692         The system provides thread-safe versions of all of the POSIX.1 functions.
- 11693         This option is required if the Threads option is supported. This is a separate option because
- 11694         thread-safe functions are useful in implementations providing other mechanisms for
- 11695         concurrency. It should only be required if needed, but it should never be prohibited.
- 11696         XSI-conformant systems support this option.
- 11697     \_POSIX\_THREAD\_SPORADIC\_SERVER
- 11698         The system supports the thread sporadic server scheduling policy.
- 11699         Support for this option provides applications with a new scheduling policy for scheduling
- 11700         aperiodic threads in hard realtime applications.
- 11701     \_POSIX\_TIMEOUTS
- 11702         The system provides timeouts for some blocking services.
- 11703         This option was created to provide a timeout capability to system services, thus allowing
- 11704         applications to include better error detection, and recovery capabilities.
- 11705     \_POSIX\_TIMERS
- 11706         The system provides higher resolution clocks with multiple timers per process.
- 11707         This option was created to support historical systems that did not provide the features. This
- 11708         option is appropriate for applications requiring higher resolution timestamps or needing to
- 11709         control the timing of multiple activities. It should only be required if needed, but it should
- 11710         never be prohibited.
- 11711     \_POSIX\_TRACE
- 11712         The system supports the Trace option.

- 11713 This option was created to allow applications to perform tracing.
- 11714 `_POSIX_TRACE_EVENT_FILTER`
- 11715 The system supports the Trace Event Filter option.
- 11716 This option is dependent on support of the Trace option.
- 11717 `_POSIX_TRACE_INHERIT`
- 11718 The system supports the Trace Inherit option.
- 11719 This option is dependent on support of the Trace option.
- 11720 `_POSIX_TRACE_LOG`
- 11721 The system supports the Trace Log option.
- 11722 This option is dependent on support of the Trace option.
- 11723 `_POSIX_TYPED_MEMORY_OBJECTS`
- 11724 The system supports the Typed Memory Objects option.
- 11725 This option was created to allow realtime applications to access different kinds of physical
- 11726 memory, and allow processes in these applications to share portions of this memory.

### 11727 **D.3.5 Configurable Limits**

- 11728 In general, the configurable limits in the `<limits.h>` header defined in the Base Definitions
- 11729 volume of IEEE Std 1003.1-2001 have been set to minimal values; many applications or
- 11730 implementations may require larger values. No profile can cite lower values.
- 11731 `{AIO_LISTIO_MAX}`
- 11732 The current minimum is likely to be inadequate for most applications. It is expected that
- 11733 this value will be increased by profiles requiring support for list input and output
- 11734 operations.
- 11735 `{AIO_MAX}`
- 11736 The current minimum is likely to be inadequate for most applications. It is expected that
- 11737 this value will be increased by profiles requiring support for asynchronous input and
- 11738 output operations.
- 11739 `{AIO_PRIO_DELTA_MAX}`
- 11740 The functionality associated with this limit is needed only by sophisticated applications. It
- 11741 is not expected that this limit would need to be increased under a general-purpose profile.
- 11742 `{ARG_MAX}`
- 11743 The current minimum is likely to need to be increased for profiles, particularly as larger
- 11744 amounts of information are passed through the environment. Many implementations are
- 11745 believed to support larger values.
- 11746 `{CHILD_MAX}`
- 11747 The current minimum is suitable only for systems where a single user is not running
- 11748 applications in parallel. It is significantly too low for any system also requiring windows,
- 11749 and if `_POSIX_JOB_CONTROL` is specified, it should be raised.
- 11750 `{CLOCKRES_MIN}`
- 11751 It is expected that profiles will require a finer granularity clock, perhaps as fine as 1  $\mu$ s,
- 11752 represented by a value of 1 000 for this limit.
- 11753 `{DELAYTIMER_MAX}`
- 11754 It is believed that most implementations will provide larger values.

|       |                                                                                                 |
|-------|-------------------------------------------------------------------------------------------------|
| 11755 | {LINK_MAX}                                                                                      |
| 11756 | For most applications and usage, the current minimum is adequate. Many implementations          |
| 11757 | have a much larger value, but this should not be used as a basis for raising the value unless   |
| 11758 | the applications to be used require it.                                                         |
| 11759 | {LOGIN_NAME_MAX}                                                                                |
| 11760 | This is not actually a limit, but an implementation parameter. No profile should impose a       |
| 11761 | requirement on this value.                                                                      |
| 11762 | {MAX_CANON}                                                                                     |
| 11763 | For most purposes, the current minimum is adequate. Unless high-speed burst serial              |
| 11764 | devices are used, it should be left as is.                                                      |
| 11765 | {MAX_INPUT}                                                                                     |
| 11766 | See {MAX_CANON}.                                                                                |
| 11767 | {MQ_OPEN_MAX}                                                                                   |
| 11768 | The current minimum should be adequate for most profiles.                                       |
| 11769 | {MQ_PRIO_MAX}                                                                                   |
| 11770 | The current minimum corresponds to the required number of process scheduling priorities.        |
| 11771 | Many realtime practitioners believe that the number of message priority levels ought to be      |
| 11772 | the same as the number of execution scheduling priorities.                                      |
| 11773 | {NAME_MAX}                                                                                      |
| 11774 | Many implementations now support larger values, and many applications and users                 |
| 11775 | assume that larger names can be used. Many existing profiles also specify a larger value.       |
| 11776 | Specifying this value will reduce the number of conforming implementations, although this       |
| 11777 | might not be a significant consideration over time. Values greater than 255 should not be       |
| 11778 | required.                                                                                       |
| 11779 | {NGROUPS_MAX}                                                                                   |
| 11780 | The value selected will typically be 8 or larger.                                               |
| 11781 | {OPEN_MAX}                                                                                      |
| 11782 | The historically common value for this has been 20. Many implementations support larger         |
| 11783 | values. If applications that use larger values are anticipated, an appropriate value should be  |
| 11784 | specified.                                                                                      |
| 11785 | {PAGESIZE}                                                                                      |
| 11786 | This is not actually a limit, but an implementation parameter. No profile should impose a       |
| 11787 | requirement on this value.                                                                      |
| 11788 | {PATH_MAX}                                                                                      |
| 11789 | Historically, the minimum has been either 1024 or indefinite, depending on the                  |
| 11790 | implementation. Few applications actually require values larger than 256, but some users        |
| 11791 | may create file hierarchies that must be accessed with longer paths. This value should only     |
| 11792 | be changed if there is a clear requirement.                                                     |
| 11793 | {PIPE_BUF}                                                                                      |
| 11794 | The current minimum is adequate for most applications. Historically, it has been larger. If     |
| 11795 | applications that write single transactions larger than this are anticipated, it should be      |
| 11796 | increased. Applications that write lines of text larger than this probably do not need it       |
| 11797 | increased, as the text line is delimited by a <newline>.                                        |
| 11798 | {POSIX_VERSION}                                                                                 |
| 11799 | This is actually not a limit, but a standard version stamp. Generally, a profile should specify |
| 11800 | IEEE Std 1003.1-2001 by a name in the normative references section, not this value.             |

|       |                                                                                               |
|-------|-----------------------------------------------------------------------------------------------|
| 11801 | {PTHREAD_DESTRUCTOR_ITERATIONS}                                                               |
| 11802 | It is unlikely that applications will need larger values to avoid loss of memory resources.   |
| 11803 | {PTHREAD_KEYS_MAX}                                                                            |
| 11804 | The current value should be adequate for most profiles.                                       |
| 11805 | {PTHREAD_STACK_MIN}                                                                           |
| 11806 | This should not be treated as an actual limit, but as an implementation parameter. No         |
| 11807 | profile should impose a requirement on this value.                                            |
| 11808 | {PTHREAD_THREADS_MAX}                                                                         |
| 11809 | It is believed that most implementations will provide larger values.                          |
| 11810 | {RTSIG_MAX}                                                                                   |
| 11811 | The current limit was chosen so that the set of POSIX.1 signal numbers can fit within a 32-   |
| 11812 | bit field. It is recognized that most existing implementations define many more signals than  |
| 11813 | are specified in POSIX.1 and, in fact, many implementations have already exceeded 32          |
| 11814 | signals (including the “null signal”). Support of {_POSIX_RTSIG_MAX} additional signals       |
| 11815 | may push some implementations over the single 32-bit word line, but is unlikely to push       |
| 11816 | any implementations that are already over that line beyond the 64 signal line.                |
| 11817 | {SEM_NSEMS_MAX}                                                                               |
| 11818 | The current value should be adequate for most profiles.                                       |
| 11819 | {SEM_VALUE_MAX}                                                                               |
| 11820 | The current value should be adequate for most profiles.                                       |
| 11821 | {SSIZE_MAX}                                                                                   |
| 11822 | This limit reflects fundamental hardware characteristics (the size of an integer), and should |
| 11823 | not be specified unless it is clearly required. Extreme care should be taken to assure that   |
| 11824 | any value that might be specified does not unnecessarily eliminate implementations            |
| 11825 | because of accidents of hardware design.                                                      |
| 11826 | {STREAM_MAX}                                                                                  |
| 11827 | This limit is very closely related to {OPEN_MAX}. It should never be larger than              |
| 11828 | {OPEN_MAX}, but could reasonably be smaller for application areas where most files are        |
| 11829 | not accessed through <i>stdio</i> . Some implementations may limit {STREAM_MAX} to 20 but     |
| 11830 | allow {OPEN_MAX} to be considerably larger. Such implementations should be allowed for        |
| 11831 | if the applications permit.                                                                   |
| 11832 | {TIMER_MAX}                                                                                   |
| 11833 | The current limit should be adequate for most profiles, but it may need to be larger for      |
| 11834 | applications with a large number of asynchronous operations.                                  |
| 11835 | {TTY_NAME_MAX}                                                                                |
| 11836 | This is not actually a limit, but an implementation parameter. No profile should impose a     |
| 11837 | requirement on this value.                                                                    |
| 11838 | {TZNAME_MAX}                                                                                  |
| 11839 | The minimum has been historically adequate, but if longer timezone names are anticipated      |
| 11840 | (particularly such values as UTC−1), this should be increased.                                |

**11841 D.3.6 Optional Behavior**

11842 In IEEE Std 1003.1-2001, there are no instances of the terms unspecified, undefined,  
11843 implementation-defined, or with the verbs “may” or “need not”, that the developers of  
11844 IEEE Std 1003.1-2001 anticipate or sanction as suitable for profile or test method citation. All of  
11845 these are merely warnings to conforming applications to avoid certain areas that can vary from  
11846 system to system, and even over time on the same system. In many cases, these terms are used  
11847 explicitly to support extensions, but profiles should not anticipate and require such extensions;  
11848 future versions of IEEE Std 1003.1 may do so.



11849 / *Rationale (Informative)*

11850 **Part E:**

11851 **Subprofiling Considerations**

11852 *The Open Group*

11853 *The Institute of Electrical and Electronics Engineers, Inc.*





|       |                                                                                                                                                                                                         |
|-------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 11897 | <i>tanh()</i> , <i>tanhf()</i> , <i>tanhf_l()</i> , <i>tanl()</i> , <i>tgamma()</i> , <i>tgammaf()</i> , <i>tgamma_l()</i> , <i>trunc()</i> , <i>truncf()</i> , <i>truncf_l()</i>                       |
| 11898 | POSIX_C_LANG_SUPPORT: General ISO C Library                                                                                                                                                             |
| 11899 | <i>abs()</i> , <i>asctime()</i> , <i>atof()</i> , <i>atoi()</i> , <i>atol()</i> , <i>atoll()</i> , <i>bsearch()</i> , <i>calloc()</i> , <i>ctime()</i> , <i>difftime()</i> , <i>div()</i> ,             |
| 11900 | <i>feclearexcept()</i> , <i>fegetenv()</i> , <i>fegetexceptflag()</i> , <i>fegetround()</i> , <i>fehldexcept()</i> , <i>feraiseexcept()</i> ,                                                           |
| 11901 | <i>fesetenv()</i> , <i>fesetexceptflag()</i> , <i>fesetround()</i> , <i>fetestexcept()</i> , <i>feupdateenv()</i> , <i>free()</i> , <i>gmtime()</i> ,                                                   |
| 11902 | <i>imaxabs()</i> , <i>imaxdiv()</i> , <i>isalnum()</i> , <i>isalpha()</i> , <i>isblank()</i> , <i>iscntrl()</i> , <i>isdigit()</i> , <i>isgraph()</i> , <i>islower()</i> ,                              |
| 11903 | <i>isprint()</i> , <i>ispunct()</i> , <i>isspace()</i> , <i>isupper()</i> , <i>isxdigit()</i> , <i>labs()</i> , <i>ldiv()</i> , <i>llabs()</i> , <i>lldiv()</i> , <i>localeconv()</i> ,                 |
| 11904 | <i>localtime()</i> , <i>malloc()</i> , <i>memchr()</i> , <i>memcmp()</i> , <i>memcpy()</i> , <i>memmove()</i> , <i>memset()</i> , <i>mktime()</i> ,                                                     |
| 11905 | <i>qsort()</i> , <i>rand()</i> , <i>realloc()</i> , <i>setlocale()</i> , <i>snprintf()</i> , <i>sprintf()</i> , <i>srand()</i> , <i>sscanf()</i> , <i>strcat()</i> , <i>strchr()</i> ,                  |
| 11906 | <i>strcmp()</i> , <i>strcoll()</i> , <i>strcpy()</i> , <i>strcspn()</i> , <i>strerror()</i> , <i>strftime()</i> , <i>strlen()</i> , <i>strncat()</i> , <i>strncpy()</i> ,                               |
| 11907 | <i>strncpy()</i> , <i>strpbrk()</i> , <i>strrchr()</i> , <i>strspn()</i> , <i>strstr()</i> , <i>strtod()</i> , <i>strtof()</i> , <i>strtoimax()</i> , <i>strtok()</i> , <i>strtol()</i> ,               |
| 11908 | <i>strtold()</i> , <i>strtoll()</i> , <i>strtol()</i> , <i>strtol()</i> , <i>strtol()</i> , <i>strtol()</i> , <i>strtol()</i> , <i>strtol()</i> , <i>strtol()</i> , <i>strtol()</i> , <i>strtol()</i> , |
| 11909 | <i>tzname</i> , <i>tzset()</i> , <i>va_arg()</i> , <i>va_copy()</i> , <i>va_end()</i> , <i>va_start()</i> , <i>vsnprintf()</i> , <i>vsprintf()</i> , <i>vsscanf()</i>                                   |
| 11910 | POSIX_C_LANG_SUPPORT_R: Thread-Safe General ISO C Library                                                                                                                                               |
| 11911 | <i>asctime_r()</i> , <i>ctime_r()</i> , <i>gmtime_r()</i> , <i>localtime_r()</i> , <i>rand_r()</i> , <i>strerror_r()</i> , <i>strtok_r()</i>                                                            |
| 11912 | POSIX_C_LANG_WIDE_CHAR: Wide-Character ISO C Library                                                                                                                                                    |
| 11913 | <i>btowc()</i> , <i>iswalnum()</i> , <i>iswalnum()</i> , <i>iswalpha()</i> , <i>iswblank()</i> , <i>iswcntrl()</i> , <i>iswctype()</i> , <i>iswdigit()</i> , <i>iswgraph()</i> ,                        |
| 11914 | <i>iswlower()</i> , <i>iswprint()</i> , <i>iswpunct()</i> , <i>iswspace()</i> , <i>iswupper()</i> , <i>iswxdigit()</i> , <i>mben()</i> , <i>mbrlen()</i> ,                                              |
| 11915 | <i>mbtowc()</i> , <i>mbsinit()</i> , <i>mbsrtowcs()</i> , <i>mbstowcs()</i> , <i>mbtowc()</i> , <i>swprintf()</i> , <i>swscanf()</i> , <i>towctrans()</i> ,                                             |
| 11916 | <i>towlower()</i> , <i>towupper()</i> , <i>vswprintf()</i> , <i>vswscanf()</i> , <i>wcrtomb()</i> , <i>wcscat()</i> , <i>wcschr()</i> , <i>wcscmp()</i> ,                                               |
| 11917 | <i>wscoll()</i> , <i>wscpy()</i> , <i>wcscspn()</i> , <i>wcsftime()</i> , <i>wcslen()</i> , <i>wcsncat()</i> , <i>wcsncmp()</i> , <i>wcsncpy()</i> ,                                                    |
| 11918 | <i>wcspbrk()</i> , <i>wcsrchr()</i> , <i>wcrtombs()</i> , <i>wcspn()</i> , <i>wcsstr()</i> , <i>wcstod()</i> , <i>wcstof()</i> , <i>wcstoimax()</i> ,                                                   |
| 11919 | <i>wcstok()</i> , <i>wcstol()</i> , <i>wcstold()</i> , <i>wcstoll()</i> , <i>wcstombs()</i> , <i>wcstoul()</i> , <i>wcstoull()</i> , <i>wcstoumax()</i> ,                                               |
| 11920 | <i>wcsxfrm()</i> , <i>wctob()</i> , <i>wctomb()</i> , <i>wctrans()</i> , <i>wctype()</i> , <i>wmemchr()</i> , <i>wmemcmp()</i> , <i>wmemcpy()</i> ,                                                     |
| 11921 | <i>wmemmove()</i> , <i>wmemset()</i>                                                                                                                                                                    |
| 11922 | POSIX_C_LIB_EXT: General C Library Extension                                                                                                                                                            |
| 11923 | <i>fnmatch()</i> , <i>getopt()</i> , <i>optarg</i> , <i>opterr</i> , <i>optind</i> , <i>optopt</i>                                                                                                      |
| 11924 | POSIX_DEVICE_IO: Device Input and Output                                                                                                                                                                |
| 11925 | <i>FD_CLR()</i> , <i>FD_ISSET()</i> , <i>FD_SET()</i> , <i>FD_ZERO()</i> , <i>clearerr()</i> , <i>close()</i> , <i>fclose()</i> , <i>fdopen()</i> , <i>feof()</i> ,                                     |
| 11926 | <i>ferror()</i> , <i>fflush()</i> , <i>fgetc()</i> , <i>fgets()</i> , <i>fileno()</i> , <i>fopen()</i> , <i>fprintf()</i> , <i>fputc()</i> , <i>fputs()</i> , <i>fread()</i> , <i>freopen()</i> ,       |
| 11927 | <i>fscanf()</i> , <i>fwrite()</i> , <i>getc()</i> , <i>getchar()</i> , <i>gets()</i> , <i>open()</i> , <i>perror()</i> , <i>printf()</i> , <i>pselect()</i> , <i>putc()</i> , <i>putchar()</i> ,        |
| 11928 | <i>puts()</i> , <i>read()</i> , <i>scanf()</i> , <i>select()</i> , <i>setbuf()</i> , <i>setvbuf()</i> , <i>stderr</i> , <i>stdin</i> , <i>stdout</i> , <i>ungetc()</i> , <i>vfprintf()</i> ,            |
| 11929 | <i>vfsanf()</i> , <i>vprintf()</i> , <i>vscanf()</i> , <i>write()</i>                                                                                                                                   |
| 11930 | POSIX_DEVICE_SPECIFIC: General Terminal                                                                                                                                                                 |
| 11931 | <i>cfgetispeed()</i> , <i>cfgetospeed()</i> , <i>cfsetispeed()</i> , <i>cfsetospeed()</i> , <i>ctermid()</i> , <i>isatty()</i> , <i>tcdrain()</i> , <i>tcflow()</i> ,                                   |
| 11932 | <i>tcflush()</i> , <i>tcgetattr()</i> , <i>tcsetattr()</i> , <i>tcsetattr()</i> , <i>tcsetattr()</i>                                                                                                    |
| 11933 | POSIX_DEVICE_SPECIFIC_R: Thread-Safe General Terminal                                                                                                                                                   |
| 11934 | <i>ttyname_r()</i>                                                                                                                                                                                      |
| 11935 | POSIX_FD_MGMT: File Descriptor Management                                                                                                                                                               |
| 11936 | <i>dup()</i> , <i>dup2()</i> , <i>fcntl()</i> , <i>fgetpos()</i> , <i>fseek()</i> , <i>fseeko()</i> , <i>fsetpos()</i> , <i>ftell()</i> , <i>ftello()</i> , <i>ftruncate()</i> , <i>lseek()</i> ,       |
| 11937 | <i>rewind()</i>                                                                                                                                                                                         |
| 11938 | POSIX_FIFO: FIFO                                                                                                                                                                                        |
| 11939 | <i>mkfifo()</i>                                                                                                                                                                                         |
| 11940 | POSIX_FILE_ATTRIBUTES: File Attributes                                                                                                                                                                  |
| 11941 | <i>chmod()</i> , <i>chown()</i> , <i>fchmod()</i> , <i>fchown()</i> , <i>umask()</i>                                                                                                                    |
| 11942 | POSIX_FILE_LOCKING: Thread-Safe Stdio Locking                                                                                                                                                           |
| 11943 | <i>flockfile()</i> , <i>ftrylockfile()</i> , <i>funlockfile()</i> , <i>getc_unlocked()</i> , <i>getchar_unlocked()</i> , <i>putc_unlocked()</i> ,                                                       |

|       |                                                                                                                 |
|-------|-----------------------------------------------------------------------------------------------------------------|
| 11944 | <i>putchar_unlocked()</i>                                                                                       |
| 11945 | POSIX_FILE_SYSTEM: File System                                                                                  |
| 11946 | <i>access(), chdir(), closedir(), creat(), fpathconf(), fstat(), getcwd(), link(), mkdir(), opendir(),</i>      |
| 11947 | <i>pathconf(), readdir(), remove(), rename(), rewinddir(), rmdir(), stat(), tmpfile(), tmpnam(),</i>            |
| 11948 | <i>unlink(), utime()</i>                                                                                        |
| 11949 | POSIX_FILE_SYSTEM_EXT: File System Extensions                                                                   |
| 11950 | <i>glob(), globfree()</i>                                                                                       |
| 11951 | POSIX_FILE_SYSTEM_R: Thread-Safe File System                                                                    |
| 11952 | <i>readdir_r()</i>                                                                                              |
| 11953 | POSIX_JOB_CONTROL: Job Control                                                                                  |
| 11954 | <i>setpgid(), tcgetpgrp(), tcsetpgrp()</i>                                                                      |
| 11955 | POSIX_MULTI_PROCESS: Multiple Processes                                                                         |
| 11956 | <i>_Exit(), _exit(), assert(), atexit(), clock(), execl(), execle(), execlp(), execv(), execve(), execvp(),</i> |
| 11957 | <i>exit(), fork(), getpgrp(), getpid(), getppid(), setsid(), sleep(), times(), wait(), waitpid()</i>            |
| 11958 | POSIX_NETWORKING: Networking                                                                                    |
| 11959 | <i>accept(), bind(), connect(), endhostent(), endnetent(), endprotoent(), endservent(),</i>                     |
| 11960 | <i>freeaddrinfo(), gai_strerror(), getaddrinfo(), gethostbyaddr(), gethostbyname(), gethostent(),</i>           |
| 11961 | <i>gethostname(), getnameinfo(), getnetbyaddr(), getnetbyname(), getnetent(), getpeername(),</i>                |
| 11962 | <i>getprotobyname(), getprotobynumber(), getprotoent(), getservbyname(), getservbyport(),</i>                   |
| 11963 | <i>getservent(), getsockname(), getsockopt(), h_errno, htonl(), htons(), if_freenameindex(),</i>                |
| 11964 | <i>if_indextoname(), if_nameindex(), if_nametoindex(), inet_addr(), inet_ntoa(), inet_ntop(),</i>               |
| 11965 | <i>inet_pton(), listen(), ntohl(), ntohs(), recv(), recvfrom(), recvmsg(), send(), sendmsg(), sendto(),</i>     |
| 11966 | <i>sethostent(), setnetent(), setprotoent(), setservent(), setsockopt(), shutdown(), socket(),</i>              |
| 11967 | <i>socketatmark(), socketpair()</i>                                                                             |
| 11968 | POSIX_PIPE: Pipe                                                                                                |
| 11969 | <i>pipe()</i>                                                                                                   |
| 11970 | POSIX_REGEX: Regular Expressions                                                                                |
| 11971 | <i>regcomp(), regerror(), regexec(), regfree()</i>                                                              |
| 11972 | POSIX_SHELL_FUNC: Shell and Utilities                                                                           |
| 11973 | <i>pclose(), popen(), system(), wordexp(), wordfree()</i>                                                       |
| 11974 | POSIX_SIGNALS: Signal                                                                                           |
| 11975 | <i>abort(), alarm(), kill(), pause(), raise(), sigaction(), sigaddset(), sigdelset(), sigemptyset(),</i>        |
| 11976 | <i>sigfillset(), sigismember(), signal(), sigpending(), sigprocmask(), sigsuspend(), sigwait()</i>              |
| 11977 | POSIX_SIGNAL_JUMP: Signal Jump Functions                                                                        |
| 11978 | <i>siglongjmp(), sigsetjmp()</i>                                                                                |
| 11979 | POSIX_SINGLE_PROCESS: Single Process                                                                            |
| 11980 | <i>confstr(), environ, errno, getenv(), setenv(), sysconf(), uname(), unsetenv()</i>                            |
| 11981 | POSIX_SYMBOLIC_LINKS: Symbolic Links                                                                            |
| 11982 | <i>lstat(), readlink(), symlink()</i>                                                                           |
| 11983 | POSIX_SYSTEM_DATABASE: System Database                                                                          |
| 11984 | <i>getgrgid(), getgrnam(), getpwnam(), getpwuid()</i>                                                           |
| 11985 | POSIX_SYSTEM_DATABASE_R: Thread-Safe System Database                                                            |
| 11986 | <i>getgrgid_r(), getgrnam_r(), getpwnam_r(), getpwuid_r()</i>                                                   |

|       |                                                                                                           |
|-------|-----------------------------------------------------------------------------------------------------------|
| 11987 | POSIX_USER_GROUPS: User and Group                                                                         |
| 11988 | <i>getegid(), geteuid(), getgid(), getgroups(), getlogin(), getuid(), setegid(), seteuid(), setgid(),</i> |
| 11989 | <i>setuid()</i>                                                                                           |
| 11990 | POSIX_USER_GROUPS_R: Thread-Safe User and Group                                                           |
| 11991 | <i>getlogin_r()</i>                                                                                       |
| 11992 | POSIX_WIDE_CHAR_DEVICE_IO: Device Input and Output                                                        |
| 11993 | <i>fgetwc(), fgetws(), fputwc(), fputws(), fwide(), fwprintf(), fwscanf(), getwc(), getwchar(),</i>       |
| 11994 | <i>putwc(), putwchar(), ungetwc(), vfwprintf(), vfwscanf(), vwprintf(), vwscanf(), wprintf(),</i>         |
| 11995 | <i>wscanf()</i>                                                                                           |
| 11996 | XSI_C_LANG_SUPPORT: XSI General C Library                                                                 |
| 11997 | <i>_tolower(), _toupper(), a64l(), daylight(), drand48(), erand48(), ffs(), getcontext(), getdate(),</i>  |
| 11998 | <i>getsubopt(), hcreate(), hdestroy(), hsearch(), iconv(), iconv_close(), iconv_open(), initstate(),</i>  |
| 11999 | <i>insque(), isascii(), jrand48(), l64a(), lcong48(), lfind(), lrand48(), lsearch(), makecontext(),</i>   |
| 12000 | <i>memccpy(), mrand48(), nrand48(), random(), remque(), seed48(), setcontext(), setstate(),</i>           |
| 12001 | <i>signgam(), srand48(), srandom(), strcasecmp(), strdup(), strfmon(), strncasecmp(), strptime(),</i>     |
| 12002 | <i>swab(), swapcontext(), tdelete(), tfind(), timezone(), toascii(), tsearch(), twalk()</i>               |
| 12003 | XSI_DBM: XSI Database Management                                                                          |
| 12004 | <i>dbm_clearerr(), dbm_close(), dbm_delete(), dbm_error(), dbm_fetch(), dbm_firstkey(),</i>               |
| 12005 | <i>dbm_nextkey(), dbm_open(), dbm_store()</i>                                                             |
| 12006 | XSI_DEVICE_IO: XSI Device Input and Output                                                                |
| 12007 | <i>fntmsg(), poll(), pread(), pwrite(), readv(), writev()</i>                                             |
| 12008 | XSI_DEVICE_SPECIFIC: XSI General Terminal                                                                 |
| 12009 | <i>grantpt(), posix_openpt(), ptsname(), unlockpt()</i>                                                   |
| 12010 | XSI_DYNAMIC_LINKING: XSI Dynamic Linking                                                                  |
| 12011 | <i>dlclose(), dlerror(), dlopen(), dlsym()</i>                                                            |
| 12012 | XSI_FD_MGMT: XSI File Descriptor Management                                                               |
| 12013 | <i>truncate()</i>                                                                                         |
| 12014 | XSI_FILE_SYSTEM: XSI File System                                                                          |
| 12015 | <i>basename(), dirname(), fchdir(), fstatvfs(), ftw(), lchown(), lockf(), mknod(), mkstemp(), nftw(),</i> |
| 12016 | <i>realpath(), seekdir(), statvfs(), sync(), telldir(), tempnam()</i>                                     |
| 12017 | XSI_I18N: XSI Internationalization                                                                        |
| 12018 | <i>catclose(), catgets(), catopen(), nl_langinfo()</i>                                                    |
| 12019 | XSI_IPC: XSI Interprocess Communication                                                                   |
| 12020 | <i>flok(), msgctl(), msgget(), msgrcv(), msgsnd(), semctl(), semget(), semop(), shmat(), shmctl(),</i>    |
| 12021 | <i>shmdt(), shmget()</i>                                                                                  |
| 12022 | XSI_JOB_CONTROL: XSI Job Control                                                                          |
| 12023 | <i>tcgetsid()</i>                                                                                         |
| 12024 | XSI_JUMP: XSI Jump Functions                                                                              |
| 12025 | <i>_longjmp(), _setjmp()</i>                                                                              |
| 12026 | XSI_MATH: XSI Maths Library                                                                               |
| 12027 | <i>j0(), j1(), jn(), scalb(), y0(), y1(), yn()</i>                                                        |
| 12028 | XSI_MULTI_PROCESS: XSI Multiple Process                                                                   |
| 12029 | <i>getpgid(), getpriority(), getrlimit(), getrusage(), getsid(), nice(), setpgrp(), setpriority(),</i>    |
| 12030 | <i>setrlimit(), ulimit(), usleep(), vfork(), waitid()</i>                                                 |

|       |                                                                                                                                                                        |
|-------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 12031 | XSI_SIGNALS: XSI Signal                                                                                                                                                |
| 12032 | <i>bsd_signal()</i> , <i>killpg()</i> , <i>sigaltstack()</i> , <i>sighold()</i> , <i>sigignore()</i> , <i>siginterrupt()</i> , <i>sigpause()</i> , <i>sigrelse()</i> , |
| 12033 | <i>sigset()</i> , <i>ualarm()</i>                                                                                                                                      |
| 12034 | XSI_SINGLE_PROCESS: XSI Single Process                                                                                                                                 |
| 12035 | <i>gethostid()</i> , <i>gettimeofday()</i> , <i>putenv()</i>                                                                                                           |
| 12036 | XSI_SYSTEM_DATABASE: XSI System Database                                                                                                                               |
| 12037 | <i>endpwent()</i> , <i>getpwent()</i> , <i>setpwent()</i>                                                                                                              |
| 12038 | XSI_SYSTEM_LOGGING: XSI System Logging                                                                                                                                 |
| 12039 | <i>closelog()</i> , <i>openlog()</i> , <i>setlogmask()</i> , <i>syslog()</i>                                                                                           |
| 12040 | XSI_THREAD_MUTEX_EXT: XSI Thread Mutex Extensions                                                                                                                      |
| 12041 | <i>pthread_mutexattr_gettype()</i> , <i>pthread_mutexattr_settype()</i>                                                                                                |
| 12042 | XSI_THREADS_EXT: XSI Threads Extensions                                                                                                                                |
| 12043 | <i>pthread_attr_getguardsize()</i> , <i>pthread_attr_getstack()</i> , <i>pthread_attr_setguardsize()</i> ,                                                             |
| 12044 | <i>pthread_attr_setstack()</i> , <i>pthread_getconcurrency()</i> , <i>pthread_setconcurrency()</i>                                                                     |
| 12045 | XSI_TIMERS: XSI Timers                                                                                                                                                 |
| 12046 | <i>getitimer()</i> , <i>setitimer()</i>                                                                                                                                |
| 12047 | XSI_USER_GROUPS: XSI User and Group                                                                                                                                    |
| 12048 | <i>endgrent()</i> , <i>endutxent()</i> , <i>getgrent()</i> , <i>getutxent()</i> , <i>getutxid()</i> , <i>getutxline()</i> , <i>pututxline()</i> ,                      |
| 12049 | <i>setgrent()</i> , <i>setregid()</i> , <i>setreuid()</i> , <i>setutxent()</i>                                                                                         |
| 12050 | XSI_WIDE_CHAR: XSI Wide-Character Library                                                                                                                              |
| 12051 | <i>wcswidth()</i> , <i>wcwidth()</i>                                                                                                                                   |





# Index

|                                        |              |
|----------------------------------------|--------------|
| /dev/tty.....                          | 22           |
| /etc/passwd .....                      | 35           |
| <pthread.h> .....                      | 163          |
| _asm_builtin_atoi().....               | 86           |
| _exit() .....                          | 103, 121     |
| _Exit() .....                          | 275          |
| _exit().....                           | 275          |
| _longjmp() .....                       | 278          |
| _POSIX_ADVISORY_INFO .....             | 284          |
| _POSIX_ASYNCHRONOUS_IO .....           | 284          |
| _POSIX_BARRIERS.....                   | 284          |
| _POSIX_CHOWN_RESTRICTED .....          | 4, 284       |
| _POSIX_CLOCK_SELECTION .....           | 284          |
| _POSIX_CPUTIME.....                    | 285          |
| _POSIX_C_SOURCE.....                   | 87, 91       |
| _POSIX_FSYNC.....                      | 285          |
| _POSIX_IPV6.....                       | 285          |
| _POSIX_JOB_CONTROL.....                | 5, 285, 289  |
| _POSIX_MAPPED_FILES.....               | 285          |
| _POSIX_MEMLOCK.....                    | 285          |
| _POSIX_MEMLOCK_RANGE .....             | 285          |
| _POSIX_MEMORY_PROTECTION .....         | 286          |
| _POSIX_MESSAGE_PASSING.....            | 286          |
| _POSIX_MONOTONIC_CLOCK.....            | 286          |
| _POSIX_NO_TRUNC .....                  | 5            |
| _POSIX_PRIORITIZED_IO .....            | 286          |
| _POSIX_PRIORITY_SCHEDULING.....        | 286          |
| _POSIX_REALTIME_SIGNALS .....          | 286          |
| _POSIX_REGEX .....                     | 286          |
| _POSIX_RTSIG_MAX .....                 | 97, 291      |
| _POSIX_SAVED_IDS.....                  | 5, 286       |
| _POSIX_SEMAPHORES .....                | 286          |
| _POSIX_SHARED_MEMORY_OBJECTS .....     | 286          |
| _POSIX_SHELL.....                      | 287          |
| _POSIX_SOURCE.....                     | 87           |
| _POSIX_SPAWN .....                     | 287          |
| _POSIX_SPINLOCKS .....                 | 287          |
| _POSIX_SPORADIC_SERVER .....           | 287          |
| _POSIX_SS_REPL_MAX.....                | 135          |
| _POSIX_SYNCHRONIZED_IO .....           | 287          |
| _POSIX_SYNC_IO .....                   | 285          |
| _POSIX_THREADS.....                    | 287          |
| _POSIX_THREAD_ATTR_STACKADDR.....      | 287          |
| _POSIX_THREAD_ATTR_STACKSIZE .....     | 287          |
| _POSIX_THREAD_PRIORITY_SCHEDULING..... | 288          |
| _POSIX_THREAD_PRIO_INHERIT .....       | 288          |
| _POSIX_THREAD_PRIO_PROTECT .....       | 288          |
| _POSIX_THREAD_PROCESS_SHARED.....      | 288          |
| _POSIX_THREAD_SAFE_FUNCTIONS.....      | 288          |
| _POSIX_THREAD_SPORADIC_SERVER .....    | 288          |
| _POSIX_TIMEOUTS.....                   | 288          |
| _POSIX_TIMERS .....                    | 288          |
| _POSIX_TRACE.....                      | 288          |
| _POSIX_TRACE_EVENT_FILTER.....         | 289          |
| _POSIX_TRACE_INHERIT.....              | 289          |
| _POSIX_TRACE_LOG.....                  | 289          |
| _POSIX_TYPED_MEMORY_OBJECTS .....      | 289          |
| _POSIX_TZNAME_MAX .....                | 59           |
| _POSIX_VDISABLE .....                  | 5            |
| _SC_PAGESIZE.....                      | 120-121      |
| _setjmp() .....                        | 278          |
| _XOPEN_SOURCE .....                    | 87           |
| __errno().....                         | 94           |
| abort() .....                          | 275          |
| access.....                            | 276          |
| access() .....                         | 35           |
| active trace stream.....               | 205          |
| adb, rationale for omission.....       | 264          |
| address families .....                 | 179          |
| addressing.....                        | 179          |
| advisory information.....              | 106          |
| aio_cancel().....                      | 115-116      |
| aio_fsync().....                       | 99, 114      |
| AIO_LISTIO_MAX.....                    | 289          |
| AIO_MAX .....                          | 289          |
| AIO_PRIO_DELTA_MAX.....                | 289          |
| aio_read().....                        | 116          |
| aio_suspend() .....                    | 114, 140     |
| aio_write() .....                      | 116          |
| alarm .....                            | 275          |
| alarm().....                           | 104, 139     |
| alias .....                            | 232, 278     |
| alias substitution.....                | 235          |
| AND lists.....                         | 251          |
| application instrumentation.....       | 193          |
| appropriate privilege .....            | 14           |
| appropriate privileges .....           | 14, 26       |
| ar .....                               | 281-282      |
| arbitrary file size .....              | 231          |
| ARG_MAX .....                          | 23, 225, 289 |
| arithmetic expansion .....             | 242          |
| as, rationale for omission.....        | 264          |

- asa .....279, 281-282
- ASCII .....25
- async-cancel safety .....177
- async-signal-safe .....103
- asynchronous error .....180
- asynchronous I/O .....114, 276
- asynchronous lists .....251
- at .....279
- atexit() .....176
- atoi() .....86
- awk .....279, 281
- background .....19-22, 70
- backslash .....233
- banner, rationale for omission .....264
- barriers .....156
- basename .....278-279
- basic regular expression .....62
- batch .....279
  - general concepts .....261
- batch environment .....258
  - option definitions .....259
- batch environment utilities
  - common behavior .....264
- batch services .....263
- batch systems
  - historical implementations .....258
  - history .....258
- bc .....278-279, 283
- BC\_BASE\_MAX .....225
- BC\_DIM\_MAX .....225
- BC\_SCALE\_MAX .....225
- bg .....232, 279
- bounded response .....277
- bracket expression
  - grammar .....66
- BRE
  - expression anchoring .....64
  - grammar lexical conventions .....66
  - matching a collating element .....62
  - matching a single character .....62
  - matching multiple characters .....64
  - ordinary character .....62
  - periods .....62
  - precedence .....64
  - special character .....62
- BSD .....16, 19, 22, 25, 69-70, 95-98, 102, 206
- built-in utilities .....231
- C Shell .....19-21
- C-language extensions .....273, 278
- c99 .....281
- calendar, rationale for omission .....264
- cancel, rationale for omission .....264
- canonical mode input processing .....70
- case .....252
- case folding .....35-36
- cat .....231
- catclose() .....280
- catgets() .....280
- catopen() .....280
- cd .....232, 279
- CEO .....63
- change history .....79, 221
- character .....14
- character encoding .....43
- character set .....43
- character set description file .....44
- character set, portable filename .....25
- character, rationale .....14
- CHARCLASS\_NAME\_MAX .....49
- CHAR\_MAX .....51
- chgrp .....231, 279-280
- CHILD\_MAX .....5, 207, 225, 251, 289
- chmod .....231, 276, 279-280
- chmod() .....27
- chown .....231, 276, 279-280
- chown() .....27
- chroot() .....26
- chroot, rationale for omission .....264
- cksum .....231, 279-280
- clock .....136
- clock tick .....15
- clock tick, rationale .....15
- CLOCKRES\_MIN .....289
- clocks .....136
- clock\_getcpuclockid() .....142, 144
- clock\_nanosleep() .....140
- CLOCK\_PROCESS\_CPUTIME\_ID .....142, 144
- CLOCK\_REALTIME .....136-140
- CLOCK\_THREAD\_CPUTIME\_ID .....142, 144
- close .....276
- close() .....120-121
- closedir .....276
- closelog() .....280
- cmp .....231, 279
- codes .....8, 85, 224
- col, rationale for omission .....264
- collating element order .....63
- COLL\_WEIGHTS\_MAX .....225
- column position .....15
- COLUMNS .....58
- comm .....279
- command .....15, 232, 278

|                                       |                                |                                     |                 |
|---------------------------------------|--------------------------------|-------------------------------------|-----------------|
| command execution.....                | 249                            | direct I/O .....                    | 16              |
| command language .....                | 273, 278                       | directory .....                     | 16              |
| command search.....                   | 249                            | directory device .....              | 67              |
| command substitution .....            | 241                            | directory entry.....                | 16              |
| compilation environment .....         | 87                             | directory files.....                | 67              |
| complex data manipulation.....        | 273, 279                       | directory protection .....          | 34              |
| compound commands.....                | 252                            | directory structure.....            | 67              |
| concurrent execution .....            | 34                             | directory, root.....                | 26              |
| conditional construct                 |                                | dirname.....                        | 278-279         |
| case .....                            | 252                            | dis, rationale for omission .....   | 265             |
| if .....                              | 253                            | display.....                        | 16              |
| configurable limits .....             | 283, 289                       | dot .....                           | 17              |
| configuration interrogation.....      | 272, 275                       | dot-dot .....                       | 17, 25, 40      |
| configuration options .....           | 281                            | double-quotes.....                  | 233             |
| shell and utilities .....             | 282                            | du .....                            | 231, 279-280    |
| system interfaces .....               | 284                            | dup.....                            | 276             |
| conformance.....                      | 5, 9, 12, 14, 37, 79, 205, 221 | dup() .....                         | 121             |
| conformance document.....             | 5                              | dup2.....                           | 276             |
| conformance document, rationale ..... | 5                              | dup2() .....                        | 121             |
| conforming application .....          | 12, 96, 228, 230               | EBUSY.....                          | 94, 163         |
| conforming application, strictly..... | 9, 12, 103                     | ECANCELED .....                     | 92              |
| confstr .....                         | 276                            | echo.....                           | 278             |
| connection indication queue .....     | 180                            | ECHOE .....                         | 73              |
| control mode.....                     | 72                             | ECHOK .....                         | 73              |
| controlling terminal.....             | 16, 69, 275                    | ECHONL.....                         | 72              |
| core .....                            | 35                             | ed.....                             | 279-280         |
| covert channel .....                  | 37                             | EDOM .....                          | 94              |
| cp .....                              | 231, 279                       | EFAULT .....                        | 92              |
| cpio, rationale for omission.....     | 264                            | effective user ID .....             | 35              |
| cpp, rationale for omission.....      | 265                            | EFTYPE .....                        | 92-93           |
| creat() .....                         | 120, 231                       | EILSEQ.....                         | 94              |
| crontab .....                         | 279                            | EINPROGRESS .....                   | 114             |
| CSIZE .....                           | 72                             | EINTR .....                         | 93, 95, 104-105 |
| csplit .....                          | 279                            | EINVAL .....                        | 93              |
| ctags.....                            | 281                            | ELOOP .....                         | 93              |
| ctime() .....                         | 277                            | emacs, rationale for omission ..... | 265             |
| cu, rationale for omission .....      | 265                            | ENAMETOOLONG .....                  | 93              |
| cut .....                             | 279                            | endgrent() .....                    | 33              |
| data access .....                     | 272, 276                       | endpwent() .....                    | 33              |
| data type.....                        | 205                            | ENOMEM.....                         | 93              |
| date .....                            | 280                            | ENOSYS .....                        | 93, 117         |
| dc, rationale for omission .....      | 265                            | ENOTSUP .....                       | 93              |
| dd.....                               | 231, 279                       | ENOTTY .....                        | 67, 92-93       |
| definitions .....                     | 85, 222                        | env .....                           | 278, 280        |
| DELAYTIMER_MAX.....                   | 289                            | environment access.....             | 272, 276        |
| determinism.....                      | 272                            | environment variable.....           | 57              |
| device number.....                    | 16                             | definition .....                    | 57              |
| device, logical.....                  | 22                             | EOVERFLOW.....                      | 93              |
| df .....                              | 231, 279-280                   | EPERM.....                          | 172             |
| diff.....                             | 279                            | EPIPE.....                          | 94              |
| dircmp, rationale for omission .....  | 265                            | Epoch.....                          | 17, 40-41, 136  |

|                                   |                        |                                     |                                               |
|-----------------------------------|------------------------|-------------------------------------|-----------------------------------------------|
| ERANGE.....                       | 94                     | field splitting.....                | 244                                           |
| ERASE.....                        | 70                     | FIFO.....                           | 17, 25, 130                                   |
| ERE.....                          | 65                     | FIFO special file.....              | 17                                            |
| alternation.....                  | 65                     | file.....                           | 17                                            |
| bracket expression.....           | 65                     | file access permissions.....        | 35                                            |
| expression anchoring.....         | 66                     | file classes.....                   | 17                                            |
| grammar.....                      | 66                     | file descriptors.....               | 105                                           |
| grammar lexical conventions.....  | 66                     | file format notation.....           | 42                                            |
| matching a collating element..... | 65                     | file hierarchy.....                 | 35                                            |
| matching a single character.....  | 65                     | file hierarchy manipulation.....    | 273, 279                                      |
| matching multiple characters..... | 65                     | file permissions.....               | 35, 69                                        |
| ordinary character.....           | 65                     | file removal.....                   | 223                                           |
| periods.....                      | 65                     | file size, arbitrary.....           | 231                                           |
| precedence.....                   | 66                     | file system.....                    | 17                                            |
| special character.....            | 65                     | file system, mounted.....           | 23                                            |
| EROFS.....                        | 94                     | file system, root.....              | 26                                            |
| errno.....                        | 91                     | file system, root of.....           | 26                                            |
| per-thread.....                   | 94                     | file times update.....              | 37                                            |
| error conditions.....             | 42                     | file, passwd.....                   | 24                                            |
| error handling.....               | 256                    | filename.....                       | 17                                            |
| error numbers.....                | 91, 95                 | filenames.....                      | 35                                            |
| errors.....                       | 246                    | fileno().....                       | 24                                            |
| escape character.....             | 233                    | filtering of trace event types..... | 202                                           |
| ex.....                           | 278-280                | filtering trace event types.....    | 202                                           |
| exec.....                         | 199                    | find.....                           | 279-280                                       |
| exec family.....                  | 19, 118, 176, 254, 275 | FIPS requirements.....              | 4                                             |
| exec, family.....                 | 226, 231               | flockfile().....                    | 95                                            |
| Execution Time Monitoring.....    | 141                    | fnmatch().....                      | 281                                           |
| execution time, measurement.....  | 37                     | fold.....                           | 279                                           |
| exit status.....                  | 246                    | fopen.....                          | 276                                           |
| exit().....                       | 103, 275               | fopen().....                        | 18, 231                                       |
| EXIT_FAILURE.....                 | 277                    | for loop.....                       | 252                                           |
| EXIT_SUCCESS.....                 | 277                    | foreground.....                     | 19-22, 68-70                                  |
| expand.....                       | 279                    | fork() ..                           | 19, 69, 110, 118, 120-121, 199, 206, 208, 275 |
| expr.....                         | 278-279                | fort77.....                         | 281                                           |
| EXPR_NEST_MAX.....                | 225                    | fpathconf.....                      | 276                                           |
| extended regular expression.....  | 65                     | fpathconf().....                    | 275                                           |
| extended security controls.....   | 35                     | fputc.....                          | 276                                           |
| false.....                        | 232, 278               | fread.....                          | 276                                           |
| fc.....                           | 232, 278               | free().....                         | 103, 175                                      |
| fchmod.....                       | 276                    | fseek.....                          | 276                                           |
| fclose.....                       | 276                    | fseeko().....                       | 206                                           |
| fcntl.....                        | 276                    | fsetpos.....                        | 276                                           |
| fcntl().....                      | 68, 93, 121, 123, 206  | fsetpos().....                      | 206                                           |
| fcntl() locks.....                | 178                    | fstat.....                          | 275-276                                       |
| fdopen().....                     | 121                    | fsync().....                        | 114                                           |
| FD_CLOEXEC.....                   | 123                    | ftello().....                       | 206                                           |
| feature test macro.....           | 87-88, 206             | ftruncate.....                      | 276                                           |
| fg.....                           | 232, 279               | ftruncate().....                    | 120, 122                                      |
| fgetc.....                        | 276                    | ftw().....                          | 231                                           |
| fgetpos().....                    | 206                    | function definition command.....    | 253                                           |

|                                             |                                |  |
|---------------------------------------------|--------------------------------|--|
| functions                                   |                                |  |
| implementation of.....                      | 86                             |  |
| use of.....                                 | 86                             |  |
| fwrite.....                                 | 276                            |  |
| gencat.....                                 | 280                            |  |
| general terminal interface.....             | 67                             |  |
| getc.....                                   | 276                            |  |
| getc().....                                 | 166                            |  |
| getch.....                                  | 276                            |  |
| getconf.....                                | 224, 275, 280-281              |  |
| getegid.....                                | 275                            |  |
| geteuid.....                                | 275                            |  |
| getgid.....                                 | 275                            |  |
| getgrent().....                             | 33                             |  |
| getgrgid.....                               | 275                            |  |
| getgrgid().....                             | 33, 167                        |  |
| getgrnam.....                               | 275                            |  |
| getgrnam().....                             | 33, 37, 167                    |  |
| getgroups().....                            | 26                             |  |
| getlogin.....                               | 275                            |  |
| getopt().....                               | 75, 280-281                    |  |
| getopts.....                                | 232, 281                       |  |
| getpgrp().....                              | 21                             |  |
| getpid.....                                 | 275                            |  |
| getpid().....                               | 105                            |  |
| getppid.....                                | 275                            |  |
| getpriority().....                          | 131                            |  |
| getpwent().....                             | 33                             |  |
| getpwnam.....                               | 275                            |  |
| getpwnam().....                             | 33, 37, 167                    |  |
| getpwuid.....                               | 275                            |  |
| getpwuid().....                             | 33, 167                        |  |
| getrlimit().....                            | 231                            |  |
| getrusage().....                            | 143                            |  |
| getty.....                                  | 69                             |  |
| getuid.....                                 | 275                            |  |
| getuid().....                               | 105, 207                       |  |
| gid_t.....                                  | 33                             |  |
| glob().....                                 | 281                            |  |
| globfree().....                             | 281                            |  |
| gmtime().....                               | 41                             |  |
| grammar conventions.....                    | 227                            |  |
| grep.....                                   | 279-280                        |  |
| group database.....                         | 18                             |  |
| group database access.....                  | 34                             |  |
| group file.....                             | 18                             |  |
| grouping commands.....                      | 252                            |  |
| head.....                                   | 279                            |  |
| headers.....                                | 76                             |  |
| here-document.....                          | 246                            |  |
| historical implementations.....             | 18                             |  |
| HOME.....                                   | 5                              |  |
| host byte order.....                        | 37                             |  |
| hosted implementation.....                  | 18                             |  |
| ICANON.....                                 | 70, 73                         |  |
| iconv().....                                | 280                            |  |
| iconv_close().....                          | 280                            |  |
| iconv_open().....                           | 280                            |  |
| id.....                                     | 280                            |  |
| if.....                                     | 253                            |  |
| implementation.....                         | 18                             |  |
| implementation, historical.....             | 18                             |  |
| implementation, hosted.....                 | 18                             |  |
| implementation, native.....                 | 23                             |  |
| implementation, specific.....               | 18                             |  |
| implementation-defined.....                 | 5-7                            |  |
| implementation-defined, rationale.....      | 5                              |  |
| incomplete pathname.....                    | 19                             |  |
| input file descriptor                       |                                |  |
| duplication.....                            | 246                            |  |
| input mode.....                             | 72                             |  |
| input processing.....                       | 70                             |  |
| canonical mode.....                         | 70                             |  |
| non-canonical mode.....                     | 71                             |  |
| inter-user communication.....               | 274, 280                       |  |
| interactive facilities.....                 | 273, 278                       |  |
| interface characteristics.....              | 68                             |  |
| interfaces.....                             | 179                            |  |
| internationalization variable.....          | 57                             |  |
| interprocess communication.....             | 105                            |  |
| invalid                                     |                                |  |
| use in RE.....                              | 61                             |  |
| ioctl().....                                | 68, 93                         |  |
| IPC.....                                    | 105                            |  |
| ISO C standard.....                         | 12, 15, 41, 68                 |  |
| .....                                       | 86-88, 90-91, 94, 96, 103, 206 |  |
| ISO/IEC 646: 1991 standard.....             | 25                             |  |
| ISTRIP.....                                 | 72                             |  |
| job control.....                            | 19-22, 24, 68-70, 96, 102, 275 |  |
| job control, implementing applications..... | 21                             |  |
| job control, implementing shells.....       | 19                             |  |
| job control, implementing systems.....      | 22                             |  |
| jobs.....                                   | 232, 279                       |  |
| join.....                                   | 279                            |  |
| kernel.....                                 | 22                             |  |
| kernel entity.....                          | 169                            |  |
| kill.....                                   | 232, 279                       |  |
| kill().....                                 | 95-97, 100, 102-103, 206       |  |
| last close.....                             | 121                            |  |
| lchown().....                               | 27                             |  |
| LC_COLLATE.....                             | 49                             |  |
| LC_CTYPE.....                               | 48                             |  |

|                                      |                        |                                    |                                      |
|--------------------------------------|------------------------|------------------------------------|--------------------------------------|
| LC_MESSAGES.....                     | 53                     | make .....                         | 281-282                              |
| LC_MONETARY .....                    | 51                     | malloc() .....                     | 103, 124-125, 154, 166-167, 175, 177 |
| LC_NUMERIC .....                     | 52, 75                 | man.....                           | 278                                  |
| LC_TIME .....                        | 52                     | map.....                           | 22                                   |
| ld, rationale for omission .....     | 265                    | mapped.....                        | 22                                   |
| legacy.....                          | 6                      | margin code notation.....          | 9                                    |
| legacy, rationale .....              | 6                      | margin codes.....                  | 8, 85, 224                           |
| lex .....                            | 281-282                | mathematical functions             |                                      |
| library routine.....                 | 22                     | error conditions.....              | 42                                   |
| limits.....                          | 224                    | NaN arguments .....                | 42                                   |
| line, rationale for omission .....   | 265                    | MAX_CANON .....                    | 70, 225, 290                         |
| LINES .....                          | 58                     | MAX_INPUT.....                     | 225, 290                             |
| LINE_MAX.....                        | 23, 32, 225            | may .....                          | 5                                    |
| link .....                           | 276                    | may, rationale .....               | 5                                    |
| link() .....                         | 16, 27                 | MCL_FUTURE.....                    | 118                                  |
| LINK_MAX .....                       | 225, 290               | MCL_INHERIT .....                  | 119                                  |
| lint, rationale for omission.....    | 265                    | memory locking.....                | 116                                  |
| lio_listio().....                    | 99, 115                | memory management .....            | 116, 276                             |
| lists.....                           | 250                    | memory management unit .....       | 117                                  |
| ln .....                             | 231, 279               | memory object.....                 | 23                                   |
| local mode .....                     | 72                     | memory synchronization.....        | 37                                   |
| locale .....                         | 46, 280                | memory-resident .....              | 22                                   |
| definition example .....             | 54                     | mesg .....                         | 280                                  |
| definition grammar .....             | 53                     | message passing .....              | 108, 276-277                         |
| grammar.....                         | 53                     | message queue .....                | 108                                  |
| lexical conventions.....             | 53                     | mkdir.....                         | 276, 279                             |
| locale configuration .....           | 273, 280               | mkfifo .....                       | 279                                  |
| locale definition.....               | 47                     | mkfifo().....                      | 18                                   |
| localedef.....                       | 280-281                | mknod().....                       | 18                                   |
| localtime() .....                    | 41, 166                | mknod, rationale for omission..... | 266                                  |
| logger .....                         | 280                    | mktime() .....                     | 41                                   |
| logical device .....                 | 22                     | mlockall() .....                   | 118                                  |
| login, rationale for omission .....  | 265                    | mmap() .....                       | 120-124                              |
| LOGIN_NAME_MAX.....                  | 290                    | MMU .....                          | 117                                  |
| LOGNAME.....                         | 5, 59                  | modem disconnect .....             | 71                                   |
| logname .....                        | 280                    | monotonic clock.....               | 140                                  |
| longjmp() .....                      | 93, 103, 173, 175, 278 | more .....                         | 278, 280                             |
| LONG_MAX.....                        | 74                     | mount point .....                  | 23                                   |
| LONG_MIN.....                        | 74                     | mount() .....                      | 23                                   |
| lorder, rationale for omission.....  | 265                    | mounted file system.....           | 23                                   |
| lp .....                             | 280                    | mprotect() .....                   | 120                                  |
| lpstat, rationale for omission ..... | 265                    | mq_open() .....                    | 109                                  |
| ls.....                              | 231, 279               | MQ_OPEN_MAX.....                   | 290                                  |
| lseek.....                           | 276                    | MQ_PRIO_MAX.....                   | 290                                  |
| lseek() .....                        | 114-115, 121, 165, 206 | mq_receive() .....                 | 109                                  |
| lstat.....                           | 275-276                | mq_send() .....                    | 109                                  |
| lstat().....                         | 27, 231                | mq_timedreceive().....             | 141                                  |
| lutime() .....                       | 27                     | mq_timedsend() .....               | 141                                  |
| macro.....                           | 7                      | msg*() .....                       | 105                                  |
| mail, rationale for omission.....    | 265                    | msgctl() .....                     | 106                                  |
| mailx .....                          | 278, 280               | msgget() .....                     | 106                                  |

|                                     |                      |                                   |                |
|-------------------------------------|----------------------|-----------------------------------|----------------|
| msgrcv()                            | 106                  | output processing                 | 71             |
| msgsnd()                            | 106                  | overflow conditions               | 205            |
| msync()                             | 120                  | overflow in dumping trace streams | 205            |
| multi-byte character                | 17, 70, 72           | overflow in trace streams         | 205            |
| multiple tasks                      | 273, 279             | pack, rationale for omission      | 266            |
| munmap()                            | 120, 122, 124, 127   | page                              | 24, 120, 123   |
| mutex                               | 158                  | PAGESIZE                          | 120, 164, 290  |
| mutex attributes                    |                      | parallel I/O                      | 165            |
| extended                            | 161                  | parameter expansion               | 240            |
| mutex initialization                | 172                  | parameters                        | 72, 236        |
| mv                                  | 231, 279             | parameters, positional            | 236            |
| name space                          | 88                   | parameters, special               | 236            |
| name space pollution                | 87-88                | parent directory                  | 25             |
| NAME_MAX                            | 225, 290             | passwd file                       | 24             |
| NaN arguments                       | 42                   | passwd, rationale for omission    | 266            |
| nanosleep()                         | 137, 139-140, 277    | paste                             | 279            |
| native implementation               | 23                   | patch                             | 279-281        |
| network byte order                  | 37                   | PATH                              | 59             |
| newgrp                              | 280                  | pathchk                           | 279            |
| news, rationale for omission        | 266                  | pathconf                          | 276            |
| NGROUPS_MAX                         | 4, 26, 225, 284, 290 | pathconf()                        | 18, 224, 275   |
| nice                                | 279                  | pathname expansion                | 244            |
| nice value                          | 23                   | pathname resolution               | 39             |
| nice()                              | 131                  | pathname, incomplete              | 19             |
| nl_langinfo()                       | 277                  | PATH_MAX                          | 93, 225, 290   |
| nm                                  | 281                  | pattern matching                  |                |
| noclobber option                    | 245                  | multiple character                | 257            |
| nohup                               | 279                  | notation                          | 256            |
| non-canonical mode input processing | 71                   | single character                  | 256            |
| non-printable                       | 32                   | patterns                          |                |
| normative references                | 5, 79, 221           | filename expansion                | 257            |
| NQS                                 | 259                  | pause                             | 275            |
| obsolescent                         | 6                    | pause()                           | 101, 104       |
| obsolescent, rationale              | 6                    | pax                               | 279-281        |
| od                                  | 279, 281             | pcat, rationale for omission      | 266            |
| open                                | 276                  | pclose()                          | 281            |
| open file description               | 24                   | Pending error                     | 180            |
| open file descriptors               | 246                  | per-thread errno                  | 94             |
| open()                              | 18, 69, 120-123, 231 | performance enhancements          | 272            |
| opendir                             | 276                  | pg, rationale for omission        | 266            |
| openlog()                           | 280                  | PID_MAX                           | 206            |
| OPEN_MAX                            | 5, 225, 290-291      | pipe                              | 19-20, 25, 276 |
| option definitions                  | 259                  | pipe()                            | 102            |
| optional behavior                   | 292                  | pipelines                         | 250            |
| options                             | 180                  | PIPE_BUF                          | 225, 290       |
| OR lists                            | 251                  | popen()                           | 278, 281-282   |
| orphaned process group              | 24, 102              | portability                       | 8, 85, 224     |
| output device                       | 67                   | portability codes                 | 8, 85, 224     |
| output file descriptor              |                      | portable character set            | 43             |
| duplication                         | 246                  | portable filename character set   | 25             |
| output mode                         | 72                   | positional parameters             | 236            |

|                               |               |                                                  |              |
|-------------------------------|---------------|--------------------------------------------------|--------------|
| POSIX locale .....            | 47            | posix_madvise() .....                            | 107          |
| POSIX.1 symbols .....         | 87            | POSIX_MADV_DONTNEED .....                        | 107          |
| POSIX.13 .....                | 124           | POSIX_MADV_RANDOM .....                          | 107          |
| POSIX2_BC_BASE_MAX .....      | 283           | POSIX_MADV_SEQUENTIAL .....                      | 107          |
| POSIX2_BC_DIM_MAX .....       | 283           | POSIX_MADV_WILLNEED .....                        | 107          |
| POSIX2_BC_SCALE_MAX .....     | 283           | posix_mem_offset() .....                         | 124-125      |
| POSIX2_BC_STRING_MAX .....    | 283           | POSIX_MULTI_PROCESS .....                        | 297          |
| POSIX2_CHAR_TERM .....        | 283           | POSIX_NETWORKING .....                           | 297          |
| POSIX2_COLL_WEIGHTS_MAX ..... | 283           | POSIX_PIPE .....                                 | 297          |
| POSIX2_C_BIND .....           | 226, 282      | POSIX_REC_INCR_XFER_SIZE .....                   | 107          |
| POSIX2_C_DEV .....            | 226, 282      | POSIX_REC_MAX_XFER_SIZE .....                    | 107          |
| POSIX2_EXPR_NEST_MAX .....    | 283           | POSIX_REC_MIN_XFER_SIZE .....                    | 107          |
| POSIX2_FORT_DEV .....         | 226, 282      | POSIX_REC_XFER_ALIGN .....                       | 107          |
| POSIX2_FORT_RUN .....         | 226, 282      | POSIX_REGEX .....                                | 297          |
| POSIX2_LINE_MAX .....         | 283           | POSIX_SHELL_FUNC .....                           | 297          |
| POSIX2_LOCALEDEF .....        | 226, 280, 282 | POSIX_SIGNALS .....                              | 297          |
| POSIX2_PBS .....              | 283           | POSIX_SIGNAL_JUMP .....                          | 297          |
| POSIX2_PBS_ACCOUNTING .....   | 283           | POSIX_SINGLE_PROCESS .....                       | 297          |
| POSIX2_PBS_CHECKPOINT .....   | 283           | posix_spawn() .....                              | 208, 275     |
| POSIX2_PBS_LOCATE .....       | 283           | posix_spawnp() .....                             | 208, 275     |
| POSIX2_PBS_MESSAGE .....      | 283           | POSIX_SYMBOLIC_LINKS .....                       | 297          |
| POSIX2_PBS_TRACK .....        | 283           | POSIX_SYSTEM_DATABASE .....                      | 297          |
| POSIX2_RE_DUP_MAX .....       | 284           | POSIX_SYSTEM_DATABASE_R .....                    | 297          |
| POSIX2_SW_DEV .....           | 226, 282      | posix_trace_eventid_open() .....                 | 200          |
| POSIX2_SYMLINKS .....         | 226           | POSIX_TRACE_LOOP .....                           | 205          |
| POSIX2_UPE .....              | 226, 282-283  | posix_typed_mem_get_info() .....                 | 124          |
| POSIX2_VERSION .....          | 284           | posix_typed_mem_open() .....                     | 124          |
| POSIX_ALLOC_SIZE_MIN .....    | 107           | POSIX_USER_GROUPS .....                          | 298          |
| POSIX_C_LANG_JUMP .....       | 295           | POSIX_USER_GROUPS_R .....                        | 298          |
| POSIX_C_LANG_MATH .....       | 295           | POSIX_VERSION .....                              | 290          |
| POSIX_C_LANG_SUPPORT .....    | 296           | POSIX_WIDE_CHAR_DEVICE_IO .....                  | 298          |
| POSIX_C_LANG_SUPPORT_R .....  | 296           | post-mortem filtering of trace event types ..... | 202          |
| POSIX_C_LANG_WIDE_CHAR .....  | 296           | pr .....                                         | 279-280      |
| POSIX_C_LIB_EXT .....         | 296           | pread() .....                                    | 165          |
| POSIX_DEVICE_IO .....         | 296           | printf .....                                     | 278-279      |
| POSIX_DEVICE_SPECIFIC .....   | 296           | printing .....                                   | 274          |
| POSIX_DEVICE_SPECIFIC_R ..... | 296           | privilege .....                                  | 35           |
| posix_fadvise() .....         | 107           | process group .....                              | 68           |
| POSIX_FADV_DONTNEED .....     | 107           | process group ID .....                           | 19-20, 68-69 |
| POSIX_FADV_NOREUSE .....      | 107           | process group lifetime .....                     | 69           |
| POSIX_FADV_RANDOM .....       | 107           | process group, orphaned .....                    | 24, 102      |
| POSIX_FADV_SEQUENTIAL .....   | 107           | process groups, concepts in job control .....    | 19           |
| POSIX_FADV_WILLNEED .....     | 107           | process ID reuse .....                           | 40           |
| POSIX_FD_MGMT .....           | 296           | process ID, rationale .....                      | 206          |
| POSIX_FIFO .....              | 296           | process management .....                         | 272, 275     |
| POSIX_FILE_ATTRIBUTES .....   | 296           | process scheduling .....                         | 129, 275     |
| POSIX_FILE_LOCKING .....      | 296           | prof, rationale for omission .....               | 266          |
| POSIX_FILE_SYSTEM .....       | 297           | profiling .....                                  | 281          |
| POSIX_FILE_SYSTEM_EXT .....   | 297           | programming manipulation .....                   | 195          |
| POSIX_FILE_SYSTEM_R .....     | 297           | prompting .....                                  | 238          |
| POSIX_JOB_CONTROL .....       | 297           | protocols .....                                  | 179          |



|                                       |                   |                                      |                                  |
|---------------------------------------|-------------------|--------------------------------------|----------------------------------|
| ps .....                              | 279-280           | PTHREAD_THREADS_MAX .....            | 291                              |
| pthread .....                         | 203               | putc .....                           | 276                              |
| pthread_attr_getguardsize() .....     | 165               | putc() .....                         | 166                              |
| pthread_attr_setguardsize() .....     | 165               | putchar .....                        | 276                              |
| PTHREAD_BARRIER_SERIAL_THREAD .....   | 156               | pwd .....                            | 279                              |
| pthread_barrier_wait() .....          | 157, 176          | pwrite() .....                       | 165                              |
| pthread_cond_init() .....             | 153               | queuing of waiting threads .....     | 177                              |
| pthread_cond_timedwait() .....        | 95, 140, 162, 284 | quote removal .....                  | 245                              |
| pthread_cond_wait() .....             | 95, 111, 162      | quoting .....                        | 233                              |
| pthread_create() .....                | 153-154           | rand() .....                         | 175                              |
| PTHREAD_CREATE_DETACHED .....         | 174               | RCS, rationale for omission .....    | 266                              |
| PTHREAD_DESTRUCTOR_ITERATIONS .....   | 291               | RE                                   |                                  |
| pthread_detach() .....                | 174               | grammar .....                        | 66                               |
| pthread_getconcurrency() .....        | 164               | RE bracket expression .....          | 62                               |
| pthread_getcpuclockid() .....         | 142, 144          | read .....                           | 232, 276, 278                    |
| pthread_join() .....                  | 95, 174           | read lock .....                      | 163                              |
| PTHREAD_KEYS_MAX .....                | 291               | read() .....                         | 19, 70, 93, 101-102, 104         |
| pthread_key_create() .....            | 155               | .....                                | 114-116, 119, 121, 165, 175, 207 |
| pthread_mutexattr_gettype() .....     | 162               | read-write attributes .....          | 162                              |
| pthread_mutexattr_settype() .....     | 162               | read-write locks .....               | 162                              |
| PTHREAD_MUTEX_DEFAULT .....           | 161               | readdir .....                        | 276                              |
| PTHREAD_MUTEX_ERRORCHECK .....        | 161               | reading an active trace stream ..... | 205                              |
| pthread_mutex_init() .....            | 153               | reading data .....                   | 70                               |
| pthread_mutex_lock() .....            | 95, 161, 175      | readlink .....                       | 276                              |
| PTHREAD_MUTEX_NORMAL .....            | 161               | readlink() .....                     | 27                               |
| PTHREAD_MUTEX_RECURSIVE .....         | 161               | realpath .....                       | 276                              |
| pthread_mutex_timedlock() .....       | 141               | realtime .....                       | 106                              |
| pthread_mutex_trylock() .....         | 161               | realtime signal delivery .....       | 98                               |
| pthread_mutex_unlock() .....          | 161               | realtime signal generation .....     | 98                               |
| PTHREAD_PROCESS_PRIVATE .....         | 163               | realtime signals .....               | 112                              |
| PTHREAD_PROCESS_SHARED .....          | 163               | red, rationale for omission .....    | 266                              |
| pthread_rwlockattr_destroy() .....    | 163               | redirect input .....                 | 246                              |
| pthread_rwlockattr_getpshared() ..... | 163               | redirect output .....                | 246                              |
| pthread_rwlockattr_init() .....       | 163               | redirection .....                    | 245                              |
| pthread_rwlockattr_setpshared() ..... | 163               | references .....                     | 5, 79, 221                       |
| pthread_rwlock_init() .....           | 163               | regcomp() .....                      | 281                              |
| PTHREAD_RWLOCK_INITIALIZER .....      | 163               | regerror() .....                     | 281                              |
| pthread_rwlock_rdlock() .....         | 163               | regexec() .....                      | 281                              |
| pthread_rwlock_t .....                | 163               | regfree() .....                      | 281                              |
| pthread_rwlock_tryrdlock() .....      | 163               | regular expression .....             | 60                               |
| pthread_rwlock_trywrlock() .....      | 163               | definitions .....                    | 60                               |
| pthread_rwlock_unlock() .....         | 164, 177          | general requirements .....           | 61                               |
| pthread_rwlock_wrlock() .....         | 163               | grammar .....                        | 66                               |
| pthread_self() .....                  | 155               | regular file .....                   | 25                               |
| pthread_setconcurrency() .....        | 164               | rejected utilities .....             | 264                              |
| pthread_setprio() .....               | 172               | remove() .....                       | 27                               |
| pthread_setschedparam() .....         | 172               | rename .....                         | 276                              |
| pthread_setspecific() .....           | 155               | rename() .....                       | 27                               |
| pthread_spin_lock() .....             | 157, 176          | renice .....                         | 279                              |
| pthread_spin_trylock() .....          | 157               | replenishment period .....           | 133                              |
| PTHREAD_STACK_MIN .....               | 291               | reserved words .....                 | 236                              |

|                                       |                        |                                   |                 |
|---------------------------------------|------------------------|-----------------------------------|-----------------|
| rewinddir.....                        | 276                    | setpriority() .....               | 131             |
| RE_DUP_MAX.....                       | 226                    | setpwent().....                   | 33              |
| rm.....                               | 231, 279               | setrlimit() .....                 | 231             |
| rmdir .....                           | 276, 279               | setsid() .....                    | 68              |
| rmdir() .....                         | 27, 94                 | setuid() .....                    | 26              |
| root directory.....                   | 26, 40                 | sh.....                           | 280, 287        |
| root file system.....                 | 26                     | shall.....                        | 6               |
| root of a file system.....            | 26                     | shall, rationale .....            | 6               |
| routing .....                         | 179                    | shar, rationale for omission..... | 266             |
| rsh, rationale for omission.....      | 266                    | shared memory .....               | 121             |
| RTSIG_MAX.....                        | 291                    | shell .....                       | 19-22           |
| samefile() .....                      | 206                    | SHELL.....                        | 59              |
| SA_NOCLDSTOP.....                     | 20                     | shell .....                       | 68, 70, 96, 102 |
| SA_SIGINFO.....                       | 100-101                | shell commands .....              | 247             |
| scheduling allocation domain.....     | 170                    | shell errors.....                 | 246             |
| scheduling contention scope .....     | 170-171                | shell execution environment.....  | 236, 256        |
| scheduling documentation.....         | 171                    | shell grammar .....               | 254             |
| scheduling policy.....                | 40                     | lexical conventions.....          | 255             |
| SCHED_FIFO.....                       | 131-132, 170, 178, 277 | rules.....                        | 255             |
| SCHED_OTHER.....                      | 131                    | shell variables.....              | 237             |
| SCHED_RR.....                         | 131, 170, 178, 277     | shell, job control .....          | 19, 96, 102     |
| SCHED_SPORADIC.....                   | 277                    | shl, rationale for omission ..... | 266             |
| scope .....                           | 3, 79, 221             | shm*() .....                      | 105             |
| sdb, rationale for omission .....     | 266                    | shmctl() .....                    | 106             |
| sdiff, rationale for omission.....    | 266                    | shmdt() .....                     | 106             |
| seconds since the Epoch .....         | 40-41                  | shm_open().....                   | 121-123         |
| security considerations.....          | 14, 17, 22, 33, 35, 69 | shm_unlink().....                 | 122-123         |
| security, monolithic privileges ..... | 14                     | should.....                       | 6               |
| sed .....                             | 279-280                | should, rationale .....           | 6               |
| sem*().....                           | 105                    | SIGABRT .....                     | 31, 95          |
| semaphore.....                        | 42                     | sigaction() .....                 | 98, 100         |
| semaphores.....                       | 110, 277               | SIGBUS .....                      | 31, 96          |
| semctl().....                         | 106                    | SIGCHLD .....                     | 20, 98, 101-102 |
| semget() .....                        | 106                    | SIGCLD .....                      | 101-102         |
| semop() .....                         | 106                    | SIGCONT.....                      | 20, 98, 101-102 |
| sem_init().....                       | 110                    | SIGEMT .....                      | 96              |
| SEM_NSEMS_MAX .....                   | 291                    | SIGEV_NONE.....                   | 98              |
| sem_open() .....                      | 110                    | SIGEV_SIGNAL .....                | 98-99           |
| sem_timedwait() .....                 | 141                    | SIGFPE .....                      | 31, 96, 98, 100 |
| sem_trywait() .....                   | 95, 112                | SIGHUP .....                      | 102             |
| SEM_VALUE_MAX .....                   | 291                    | SIGILL .....                      | 31, 96          |
| sem_wait().....                       | 95, 112                | SIGINT.....                       | 21, 168         |
| sequential lists.....                 | 251                    | SIGIOT.....                       | 95-96           |
| session.....                          | 20, 24, 69             | SIGKILL.....                      | 95, 98, 102     |
| set.....                              | 238                    | siglongjmp() .....                | 93, 103, 278    |
| setgid().....                         | 26                     | signal .....                      | 26              |
| setgrent() .....                      | 33                     | signal acceptance .....           | 97              |
| setjmp() .....                        | 278                    | signal actions .....              | 101             |
| setlocale().....                      | 277                    | signal concepts.....              | 95              |
| setlogmask().....                     | 280                    | signal delivery .....             | 97              |
| setpgid().....                        | 19-21, 68-69           | signal generation.....            | 97              |

|                                     |                     |                                          |                                     |
|-------------------------------------|---------------------|------------------------------------------|-------------------------------------|
| signal names .....                  | 95                  | specific implementation .....            | 18                                  |
| signal() .....                      | 95, 98              | spell, rationale for omission .....      | 267                                 |
| signals .....                       | 180, 256            | spin locks .....                         | 157-158                             |
| SIGPIPE .....                       | 31, 94              | split .....                              | 279                                 |
| sigprocmask() .....                 | 97                  | sporadic server scheduling policy .....  | 133                                 |
| SIGRTMAX .....                      | 99, 101             | SSIZE_MAX .....                          | 207, 291                            |
| SIGRTMIN .....                      | 99, 101             | SS_REPL_MAX .....                        | 135                                 |
| SIGSEGV .....                       | 31, 96, 100         | standard I/O streams .....               | 105                                 |
| sigsetjmp() .....                   | 278                 | stat .....                               | 231, 275-276                        |
| sigset_t .....                      | 95                  | stat() .....                             | 16, 120, 231                        |
| SIGSTOP .....                       | 102                 | state-dependent character encoding ..... | 44                                  |
| sigsuspend() .....                  | 101, 104            | statvfs() .....                          | 231                                 |
| SIGSYS .....                        | 96                  | STREAMS .....                            | 105                                 |
| SIGTERM .....                       | 95                  | STREAM_MAX .....                         | 291                                 |
| sigtimedwait() .....                | 95, 114, 140        | strings .....                            | 281                                 |
| SIGTRAP .....                       | 96                  | strip .....                              | 281                                 |
| SIGTSTP .....                       | 21, 102             | structures, additions to .....           | 88                                  |
| SIGTTIN .....                       | 21, 70, 102         | stty .....                               | 58, 280                             |
| SIGTTOU .....                       | 20, 70, 102         | su, rationale for omission .....         | 267                                 |
| SIGUSR1 .....                       | 95                  | subprofiling .....                       | 11                                  |
| SIGUSR2 .....                       | 95                  | subprofiling option groups .....         | 295                                 |
| sigwait() .....                     | 95, 175             | subshells .....                          | 21                                  |
| sigwaitinfo() .....                 | 95, 114             | successfully completed .....             | 32                                  |
| sigwait_multiple() .....            | 97                  | sum .....                                | 231                                 |
| SIG_DFL .....                       | 97-98, 101          | sum, rationale for omission .....        | 267                                 |
| SIG_IGN .....                       | 20, 97-98, 101, 104 | superuser .....                          | 14, 26, 35, 238, 264                |
| simple commands .....               | 248                 | supplementary group ID .....             | 26                                  |
| single-quotes .....                 | 233                 | supplementary groups .....               | 35                                  |
| size, rationale for omission .....  | 267                 | symbolic constant .....                  | 7                                   |
| SI_USER .....                       | 100                 | symbolic link .....                      | 27                                  |
| sleep .....                         | 275, 278            | symbolic name .....                      | 7                                   |
| sleep() .....                       | 103-104, 277        | symbols .....                            | 87                                  |
| socket I/O mode .....               | 180                 | SYMLOOP_MAX .....                        | 93                                  |
| socket out-of-band data state ..... | 180                 | synchronized I/O .....                   | 115, 276                            |
| socket owner .....                  | 180                 | data integrity completion .....          | 32, 115                             |
| socket queue limit .....            | 180                 | file integrity completion .....          | 32, 115                             |
| socket receive queue .....          | 180                 | synchronously-generated signal .....     | 31                                  |
| socket types .....                  | 179                 | sysconf() .....                          | 18, 117, 120-121, 168, 224, 275-276 |
| sockets .....                       | 179                 | syslog() .....                           | 280                                 |
| Internet Protocols .....            | 180                 | system call .....                        | 31                                  |
| IPv4 .....                          | 181                 | system documentation .....               | 6                                   |
| IPv6 .....                          | 181                 | system environment .....                 | 274, 280                            |
| local UNIX connection .....         | 180                 | System III .....                         | 25, 206                             |
| software development .....          | 274, 281            | system interfaces .....                  | 208                                 |
| sort .....                          | 279-280             | system reboot .....                      | 32                                  |
| spawn example .....                 | 208                 | System V .....                           | 16, 22, 96-97, 101-102              |
| special built-in .....              | 253                 | system() .....                           | 278, 281-282                        |
| special built-in utilities .....    | 258                 | tabs .....                               | 279                                 |
| special characters .....            | 71                  | tail .....                               | 279                                 |
| special control character .....     | 73                  | talk .....                               | 278, 280                            |
| special parameters .....            | 236                 | tar, rationale for omission .....        | 267                                 |

|                             |              |                                |                      |
|-----------------------------|--------------|--------------------------------|----------------------|
| tcgetattr()                 | 20           | trace event types              | 202                  |
| tcgetpgrp()                 | 21, 68-69    | trace examples                 | 191                  |
| tcsetattr()                 | 20, 68       | trace model                    | 186                  |
| tcsetpgrp()                 | 20-21        | trace operation control        | 191                  |
| tee                         | 278          | trace storage                  | 190                  |
| TERM                        | 58           | trace stream attribute         | 196                  |
| terminal access control     | 69           | trace stream states            | 189                  |
| terminal device file        | 68           | tracing                        | 42, 181              |
| closing                     | 71           | tracing all processes          | 189                  |
| terminal type               | 67           | tracing, detailed objectives   | 182                  |
| terminology                 | 5, 85, 222   | triggering                     | 204                  |
| termios structure           | 72           | troff                          | 279                  |
| test                        | 278-279      | trojan horse                   | 14                   |
| TeX                         | 279          | true                           | 232, 278             |
| text file                   | 32           | tty                            | 280                  |
| thread                      | 33           | ttyname()                      | 166                  |
| thread cancelability states | 175          | TTY_NAME_MAX                   | 291                  |
| thread cancelability type   | 175          | typed memory                   | 124                  |
| thread concurrency level    | 164          | TZ                             | 59                   |
| thread creation attributes  | 152          | TZNAME_MAX                     | 291                  |
| thread ID                   | 33, 168      | tzset()                        | 277                  |
| thread interactions         | 179          | ualarm                         | 275                  |
| thread mutex                | 169          | UID_MAX                        | 207                  |
| thread read-write lock      | 177          | uid_t                          | 33                   |
| thread scheduling           | 169          | ULONG_MAX                      | 224                  |
| thread stack guard size     | 164          | umask                          | 232, 280             |
| thread-safe                 | 33           | umask()                        | 275                  |
| thread-safe function        | 33           | umount()                       | 23                   |
| thread-safety               | 42, 165      | unalias                        | 232, 278             |
| thread-safety, rationale    | 42           | uname                          | 280                  |
| thread-specific data        | 154          | uname()                        | 275                  |
| threads                     | 152          | unbounded priority inversion   | 172                  |
| implementation models       | 154          | undefined                      | 6                    |
| tilde expansion             | 239          | undefined, rationale           | 6                    |
| time                        | 278, 281     | unexpand                       | 279                  |
| time()                      | 92           | uniq                           | 279-280              |
| timeouts                    | 145          | unlink                         | 276                  |
| timers                      | 136          | unlink()                       | 27, 94, 120, 122-123 |
| TIMER_ABSTIME               | 137-138      | unpack, rationale for omission | 267                  |
| TIMER_MAX                   | 291          | unsafe functions               | 103                  |
| timer_settime()             | 137-138      | unspecified                    | 6                    |
| times()                     | 92, 143, 275 | unspecified, rationale         | 6                    |
| timestamp clock             | 204          | until loop                     | 253                  |
| time_t                      | 41           | user database                  | 33                   |
| token recognition           | 235          | user database access           | 34                   |
| TOSTOP                      | 20           | user requirements              | 271                  |
| touch                       | 231, 279     | usleep                         | 275                  |
| tput                        | 280          | utility                        | 42                   |
| tr                          | 279-280      | utility argument syntax        | 73                   |
| trace analyzer              | 193          | utility conventions            | 73                   |
| trace event type-filtering  | 202          | utility description defaults   | 227                  |

|                                   |                                      |                           |         |
|-----------------------------------|--------------------------------------|---------------------------|---------|
| utility limits .....              | 224                                  | XSI_MATH.....             | 298     |
| utility syntax guidelines.....    | 74                                   | XSI_MULTI_PROCESS.....    | 298     |
| utime .....                       | 276                                  | XSI_SIGNALS .....         | 299     |
| utime() .....                     | 27                                   | XSI_SINGLE_PROCESS.....   | 299     |
| uucp.....                         | 280                                  | XSI_SYSTEM_DATABASE ..... | 299     |
| uudecode .....                    | 279-280                              | XSI_SYSTEM_LOGGING .....  | 299     |
| uuencode .....                    | 279-280                              | XSI_THREADS_EXT.....      | 299     |
| variable assignment .....         | 42                                   | XSI_THREAD_MUTEX_EXT..... | 299     |
| variables .....                   | 236                                  | XSI_TIMERS .....          | 299     |
| VEOF.....                         | 73                                   | XSI_USER_GROUPS.....      | 299     |
| VEOL.....                         | 73                                   | XSI_WIDE_CHAR.....        | 299     |
| Version 7.....                    | 40, 233                              | yacc .....                | 281-282 |
| vhangup() .....                   | 22                                   |                           |         |
| vi .....                          | 278, 280                             |                           |         |
| virtual processor.....            | 34                                   |                           |         |
| VMIN .....                        | 73                                   |                           |         |
| VTIME.....                        | 73                                   |                           |         |
| wait.....                         | 232, 278                             |                           |         |
| wait().....                       | 92, 96, 101, 103-104, 275            |                           |         |
| waitpid() .....                   | 20, 24, 103, 206, 275                |                           |         |
| wall, rationale for omission..... | 267                                  |                           |         |
| wc.....                           | 279                                  |                           |         |
| WERASE.....                       | 70                                   |                           |         |
| while loop.....                   | 253                                  |                           |         |
| who.....                          | 280                                  |                           |         |
| wide-character codes .....        | 44                                   |                           |         |
| word expansions.....              | 239                                  |                           |         |
| wordexp() .....                   | 281                                  |                           |         |
| wordfree() .....                  | 281                                  |                           |         |
| write.....                        | 276, 278, 280                        |                           |         |
| write lock.....                   | 163                                  |                           |         |
| write() .....                     | 19-20, 70, 93                        |                           |         |
| .....                             | 101-102, 104, 114-116, 119, 121, 207 |                           |         |
| writing data.....                 | 71                                   |                           |         |
| WUNTRACED .....                   | 20                                   |                           |         |
| xargs .....                       | 278                                  |                           |         |
| XSI.....                          | 34                                   |                           |         |
| XSI IPC .....                     | 105                                  |                           |         |
| XSI supported functions .....     | 159                                  |                           |         |
| XSI threads extensions .....      | 160                                  |                           |         |
| XSI_C_LANG_SUPPORT .....          | 298                                  |                           |         |
| XSI_DBM.....                      | 298                                  |                           |         |
| XSI_DEVICE_IO .....               | 298                                  |                           |         |
| XSI_DEVICE_SPECIFIC.....          | 298                                  |                           |         |
| XSI_DYNAMIC_LINKING .....         | 298                                  |                           |         |
| XSI_FD_MGMT .....                 | 298                                  |                           |         |
| XSI_FILE_SYSTEM.....              | 298                                  |                           |         |
| XSI_I18N.....                     | 298                                  |                           |         |
| XSI_IPC.....                      | 298                                  |                           |         |
| XSI_JOB_CONTROL .....             | 298                                  |                           |         |
| XSI_JUMP.....                     | 298                                  |                           |         |

