

IEEE Std 1003.1-2001
(Revision of IEEE Std 1003.1-1996
and IEEE Std 1003.2-1992)

Open Group Technical Standard
Base Specifications, Issue 6

1003.1TM

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

System Interfaces, Issue 6

Approved 12 September 2001
The Open Group

IEEE Sponsor

Portable Applications Standards Committee
of the
IEEE Computer Society

Approved 6 December 2001
IEEE-SA Standards Board



THE *Open* GROUP

Abstract

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. It is the single common revision to IEEE Std 1003.1-1996, IEEE Std 1003.2-1992, and the Base Specifications of The Open Group Single UNIX[®]† Specification, Version 2. 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, containing 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[®]†), parent, shell, stream, string, synchronous, system, thread, X/Open System Interface (XSI)

† See **Trademarks** (on page xl).

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.

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 <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.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with the IEEE.¹ 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.

The IEEE and its designees are the sole entities that may authorize the use of IEEE-owned certification marks and/or trademarks to indicate compliance with the materials set forth herein. 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.² 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>.

Full catalog and ordering information on all IEEE publications is available from the IEEE Online Catalog & Store at <http://shop.ieee.org/store>.

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

2. Please refer to the special provisions for this standard on page iii 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, a vendor and technology-neutral consortium, is committed to delivering greater business efficiency by bringing together buyers and suppliers of information technology to lower the time, cost, and risks associated with integrating new technology across the enterprise.

The Open Group's mission is to offer all organizations concerned with open information infrastructures a forum to share knowledge, integrate open initiatives, and certify approved products and processes in a manner in which they continue to trust our impartiality.

In the global eCommerce world of today, no single economic entity can achieve independence while still ensuring interoperability. The assurance that products will interoperate with each other across differing systems and platforms is essential to the success of eCommerce and business workflow. The Open Group, with its proven testing and certification program, is the international guarantor of interoperability in the new century.

The Open Group provides opportunities to exchange information and shape the future of IT. The Open Group's members include some of the largest and most influential organizations in the world. The flexible structure of The Open Groups membership allows for almost any organization, no matter what their size, to join and have a voice in shaping the future of the IT world.

More information is available on The Open Group web site at <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 <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 <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 <http://www.opengroup.org/corrigenda>.

Full catalog and ordering information on all Open Group publications is available at <http://www.opengroup.org/pubs>.

Contents

Chapter 1	Introduction.....	1
1.1	Scope.....	1
1.2	Conformance	1
1.3	Normative References	1
1.4	Change History	1
1.5	Terminology.....	1
1.6	Definitions	3
1.7	Relationship to Other Formal Standards	3
1.8	Portability	3
1.8.1	Codes.....	3
1.9	Format of Entries.....	11
 Chapter 2	 General Information	 13
2.1	Use and Implementation of Functions	13
2.2	The Compilation Environment	13
2.2.1	POSIX.1 Symbols	13
2.2.1.1	The _POSIX_C_SOURCE Feature Test Macro.....	14
2.2.1.2	The _XOPEN_SOURCE Feature Test Macro.....	14
2.2.2	The Name Space.....	14
2.3	Error Numbers.....	21
2.3.1	Additional Error Numbers.....	28
2.4	Signal Concepts.....	28
2.4.1	Signal Generation and Delivery.....	28
2.4.2	Realtime Signal Generation and Delivery	29
2.4.3	Signal Actions	31
2.4.4	Signal Effects on Other Functions	34
2.5	Standard I/O Streams.....	34
2.5.1	Interaction of File Descriptors and Standard I/O Streams.....	35
2.5.2	Stream Orientation and Encoding Rules	37
2.6	STREAMS	38
2.6.1	Accessing STREAMS.....	39
2.7	XSI Interprocess Communication	39
2.7.1	IPC General Description.....	40
2.8	Realtime	41
2.8.1	Realtime Signals.....	41
2.8.2	Asynchronous I/O	41
2.8.3	Memory Management	43
2.8.3.1	Memory Locking.....	43
2.8.3.2	Memory Mapped Files.....	43
2.8.3.3	Memory Protection.....	44
2.8.3.4	Typed Memory Objects	44
2.8.4	Process Scheduling	44

2.8.5	Clocks and Timers	48
2.9	Threads.....	50
2.9.1	Thread-Safety.....	50
2.9.2	Thread IDs.....	51
2.9.3	Thread Mutexes.....	51
2.9.4	Thread Scheduling.....	52
2.9.5	Thread Cancellation	54
2.9.5.1	Cancelability States	54
2.9.5.2	Cancellation Points	55
2.9.5.3	Thread Cancellation Cleanup Handlers.....	57
2.9.5.4	Async-Cancel Safety.....	57
2.9.6	Thread Read-Write Locks.....	58
2.9.7	Thread Interactions with Regular File Operations	58
2.10	Sockets.....	58
2.10.1	Address Families.....	58
2.10.2	Addressing	59
2.10.3	Protocols	59
2.10.4	Routing.....	59
2.10.5	Interfaces.....	59
2.10.6	Socket Types.....	59
2.10.7	Socket I/O Mode.....	60
2.10.8	Socket Owner.....	60
2.10.9	Socket Queue Limits	60
2.10.10	Pending Error.....	60
2.10.11	Socket Receive Queue.....	61
2.10.12	Socket Out-of-Band Data State	61
2.10.13	Connection Indication Queue	62
2.10.14	Signals	62
2.10.15	Asynchronous Errors	62
2.10.16	Use of Options.....	63
2.10.17	Use of Sockets for Local UNIX Connections.....	66
2.10.17.1	Headers	66
2.10.18	Use of Sockets over Internet Protocols.....	66
2.10.19	Use of Sockets over Internet Protocols Based on IPv4.....	67
2.10.19.1	Headers	67
2.10.20	Use of Sockets over Internet Protocols Based on IPv6.....	67
2.10.20.1	Addressing	67
2.10.20.2	Compatibility with IPv4.....	68
2.10.20.3	Interface Identification.....	68
2.10.20.4	Options.....	69
2.10.20.5	Headers	70
2.11	Tracing.....	70
2.11.1	Tracing Data Definitions.....	71
2.11.1.1	Structures.....	71
2.11.1.2	Trace Stream Attributes.....	75
2.11.2	Trace Event Type Definitions	76
2.11.2.1	System Trace Event Type Definitions.....	76
2.11.2.2	User Trace Event Type Definitions.....	79

	2.11.3	Trace Functions.....	79
	2.12	Data Types.....	80
Chapter	3	System Interfaces	83
		Index.....	1663

List of Tables

2-1	Value of Level for Socket Options.....	63
2-2	Socket-Level Options.....	64
2-3	Trace Option: System Trace Events.....	77
2-4	Trace and Trace Event Filter Options: System Trace Events.....	78
2-5	Trace and Trace Log Options: System Trace Events.....	78
2-6	Trace, Trace Log, and Trace Event Filter Options: System Trace Events.....	79
2-7	Trace Option: User Trace Event	79

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 both an IEEE 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.³ 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 and The Open Group's Technical Standard designation, and, if approved, 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,⁴ 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 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 (this volume)
- Rationale (Informative)

This Volume

The System Interfaces volume describes the interfaces offered to application programs by POSIX-conformant systems. Readers are expected to be experienced C language programmers, and to be familiar with the Base Definitions volume.

This volume is structured as follows:

- Chapter 1 explains the status of this volume and its relationship to other formal standards.
- Chapter 2 contains important concepts, terms, and caveats relating to the rest of this volume.
- Chapter 3 defines the functional interfaces to the POSIX-conformant system.

Comprehensive references are available in the index.

Typographical Conventions

The following typographical conventions are used throughout this standard. In the text, this standard is referred to as IEEE Std 1003.1-2001, which is technically identical to The Open Group Base Specifications, Issue 6.

The typographical conventions listed here are for ease of reading only. Editorial inconsistencies in the use of typography are unintentional and have no normative meaning in this standard.

Reference	Example	Notes
C-Language Data Structure	aiocb	
C-Language Data Structure Member	<i>aio_lio_opcode</i>	
C-Language Data Type	long	
C-Language External Variable	<i>errno</i>	
C-Language Function	<i>system()</i>	
C-Language Function Argument	<i>arg1</i>	
C-Language Function Family	<i>exec</i>	
C-Language Header	<sys/stat.h>	
C-Language Keyword	return	
C-Language Macro with Argument	<i>assert()</i>	
C-Language Macro with No Argument	INET_ADDRSTRLEN	
C-Language Preprocessing Directive	#define	
Commands within a Utility	a, c	
Conversion Specification, Specifier/Modifier Character	%A, g, E	1
Environment Variable	<i>PATH</i>	
Error Number	[EINTR]	
Example Output	Hello, World	
Filename	/tmp	
Literal Character	'c', '\r', '\'	2
Literal String	"abcde"	2
Optional Items in Utility Syntax	[]	
Parameter	<i><directory pathname></i>	
Special Character	<i><newline></i>	3
Symbolic Constant	_POSIX_VDISABLE	
Symbolic Limit, Configuration Value	{LINE_MAX}	4
Syntax	#include <sys/stat.h>	

Reference	Example	Notes
User Input and Example Code	echo Hello, World	5
Utility Name	awk	
Utility Operand	file_name	
Utility Option	-c	
Utility Option with Option-Argument	-w width	

Notes:

1. Conversion specifications, specifier characters, and modifier characters are used primarily in date-related functions and utilities and the *fprintf* and *fscanf* formatting functions.
2. Unless otherwise noted, the quotes shall not be used as input or output. When used in a list item, the quotes are omitted. For literal characters, `'\'` (or any of the other sequences such as `''`) is the same as the C constant `'\\'` (or `'\''`).
3. The style selected for some of the special characters, such as <newline>, matches the form of the input given to the *localedef* utility. Generally, the characters selected for this special treatment are those that are not visually distinct, such as the control characters <tab> or <newline>.
4. Names surrounded by braces represent symbolic limits or configuration values which may be declared in appropriate headers by means of the C **#define** construct.
5. Brackets shown in this font, "[]", are part of the syntax and do *not* indicate optional items. In syntax the `'|'` symbol is used to separate alternatives, and ellipses ("...") are used to show that additional arguments are optional.

Shading is used to identify extensions and options; see Section 1.8.1 (on page 3).

Footnotes and notes within the body of the normative text are for information only (informative).

Informative sections (such as Rationale, Change History, Application Usage, and so on) are denoted by continuous shading bars in the margins.

Ranges of values are indicated with parentheses or brackets as follows:

- (a,b) means the range of all values from a to b , including neither a nor b .
- $[a,b]$ means the range of all values from a to b , including a and b .
- $[a,b)$ means the range of all values from a to b , including a , but not b .
- $(a,b]$ means the range of all values from a to b , including b , but not a .

Notes:

1. Symbolic limits are used in this volume instead of fixed values for portability. The values of most of these constants are defined in the Base Definitions volume, **<limits.h>** or **<unistd.h>**.
2. The values of errors are defined in the Base Definitions volume, **<errno.h>**.

Participants

The Austin Group

This standard 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. 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
Peter Anvin
Pierre-Jean Arcos
Jay Ashford
Bouazza Bachar
Theodore P. Baker
Robert Barned
Joel Berman
David J. Blackwood
Shirley Bockstahler-Brandt
James Bottomley
Walter Briscoe
Andries Brouwer
Mark Brown
Eric W. Burger
Alan Burns
Andries Brouwer
Dave Butenhof
Keith Chow
Geoff Clare
Donald W. Cragun
Lee Damico
Juan Antonio De La Puente
Ming De Zhou
Steven J. Dovich
Richard P. Draves
Ulrich Drepper
Paul Eggert
Philip H. Enslow
Joanna Farley
Clive D.W. Feather
Pete Forman
Mark Funkenhauser
Lois Goldthwaite
Andrew Gollan

Michael Gonzalez
Karen D. Gordon
Joseph M. Gwinn
Steven A. Haaser
Charles E. Hammons
Chris J. Harding
Barry Hedquist
Vincent E. Henley
Karl Heubaum
Jon Hitchcock
Yvette Ho Sang
Niklas Holsti
Thomas Hosmer
Cathy Hughes
Jim D. Isaak
Lowell G. Johnson
Michael B. Jones
Andrew Josey
Michael J. Karels
Michael Kavanaugh
David Korn
Steven Kramer
Thomas M. Kurihara
Marc Aurele La France
C. Douglass Locke
Nick Maclaren
Roger J. Martin
Craig H. Meyer
Jim Meyering
Gary Miller
Finnbarr P. Murphy
Joseph S. Myers
John Napier
Peter E. Obermayer
James T. Oblinger

Sandra O'Donnell
Frank Prindle
Francois Riche
John D. Riley
Andrew K. Roach
Helmut Roth
Jaideep Roy
Curtis Royster Jr.
Stephen C. Schwarm
Glen Seeds
Richard Seibel
David L. Shrods Jr.
W. Olin Sibert
Keld Jorn Simonsen
Curtis Smith
Raja Srinivasan
Nicholas Stoughton
Marc J. Teller
Donn S. Terry
Fred Tydeman
Mark-Rene Uchida
Scott A. Valcourt
Peter Van Der Veen
Michael W. Vannier
Eric Vought
Frederick N. Webb
Paul A.T. Wolfgang
Garrett Wollman
James Youngman
Oren Yuen
Janusz Zalewski
Jim Zepeda
Jason Zions

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

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[™] is a trademark of the Institute of Electrical and Electronic Engineers, Inc.

AIX[®] is a registered trademark of IBM Corporation.

AT&T[®] is a registered trademark of AT&T in the U.S.A. and other countries.

BSD[™] is a trademark of the University of California, Berkeley, U.S.A.

Hewlett-Packard[®], HP[®], and HP-UX[®] are registered trademarks of Hewlett-Packard Company.

IBM[®] is a registered trademark of International Business Machines Corporation.

Motif[®], OSF/1[®], UNIX[®], and the “X Device” are registered trademarks and IT DialTone[™] and The Open Group[™] are trademarks of The Open Group in the U.S. and other countries.

POSIX[®] is a registered trademark of the Institute of Electrical and Electronic Engineers, Inc.

Sun[®] and Sun Microsystems[®] are registered trademarks of Sun Microsystems, Inc.

/usr/group[®] 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 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 13: Latin Alphabet No. 7

Part 14: Latin Alphabet No. 8

Part 15: Latin Alphabet No. 9

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).

Referenced Documents

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.

POSIX.13:-1998

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

Referenced Documents

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).

Introduction

1.1 Scope

The scope of IEEE Std 1003.1-2001 is described in the Base Definitions volume of IEEE Std 1003.1-2001.

1.2 Conformance

Conformance requirements for IEEE Std 1003.1-2001 are defined in the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 2, Conformance.

1.3 Normative References

Normative references for IEEE Std 1003.1-2001 are defined in the Base Definitions volume of IEEE Std 1003.1-2001.

1.4 Change History

Change history is described in the Rationale (Informative) volume of IEEE Std 1003.1-2001, and in the CHANGE HISTORY section of reference pages.

1.5 Terminology

This section appears in the Base Definitions volume of IEEE Std 1003.1-2001, but is repeated here for convenience:

For the purposes of IEEE Std 1003.1-2001, the following terminology definitions apply:

can

Describes a permissible optional feature or behavior available to the user or application. The feature or behavior is mandatory for an implementation that conforms to IEEE Std 1003.1-2001. An application can rely on the existence of the feature or behavior.

implementation-defined

Describes a value or behavior that is not defined by IEEE Std 1003.1-2001 but is selected by an implementor. The value or behavior may vary among implementations that conform to IEEE Std 1003.1-2001. An application should not rely on the existence of the value or behavior. An application that relies on such a value or behavior cannot be assured to be portable across conforming implementations.

The implementor shall document such a value or behavior so that it can be used correctly by an application.

legacy

Describes a feature or behavior that is being retained for compatibility with older applications, but which has limitations which make it inappropriate for developing portable

applications. New applications should use alternative means of obtaining equivalent functionality.

may

Describes a feature or behavior that is optional for an implementation that conforms to IEEE Std 1003.1-2001. An application should not rely on the existence of the feature or behavior. An application that relies on such a feature or behavior cannot be assured to be portable across conforming implementations.

To avoid ambiguity, the opposite of *may* is expressed as *need not*, instead of *may not*.

shall

For an implementation that conforms to IEEE Std 1003.1-2001, describes a feature or behavior that is mandatory. An application can rely on the existence of the feature or behavior.

For an application or user, describes a behavior that is mandatory.

should

For an implementation that conforms to IEEE Std 1003.1-2001, describes a feature or behavior that is recommended but not mandatory. An application should not rely on the existence of the feature or behavior. An application that relies on such a feature or behavior cannot be assured to be portable across conforming implementations.

For an application, describes a feature or behavior that is recommended programming practice for optimum portability.

undefined

Describes the nature of a value or behavior not defined by IEEE Std 1003.1-2001 which results from use of an invalid program construct or invalid data input.

The value or behavior may vary among implementations that conform to IEEE Std 1003.1-2001. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.

unspecified

Describes the nature of a value or behavior not specified by IEEE Std 1003.1-2001 which results from use of a valid program construct or valid data input.

The value or behavior may vary among implementations that conform to IEEE Std 1003.1-2001. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.

1.6 Definitions

Concepts and definitions are defined in the Base Definitions volume of IEEE Std 1003.1-2001.

1.7 Relationship to Other Formal Standards

Great care has been taken to ensure that this volume of IEEE Std 1003.1-2001 is fully aligned with the following standards:

ISO C (1999)

ISO/IEC 9899: 1999, Programming Languages — C.

Parts of the ISO/IEC 9899:1999 standard (hereinafter referred to as the ISO C standard) are referenced to describe requirements also mandated by this volume of IEEE Std 1003.1-2001. Some functions and headers included within this volume of IEEE Std 1003.1-2001 have a version in the ISO C standard; in this case CX markings are added as appropriate to show where the ISO C standard has been extended (see Section 1.8.1). Any conflict between this volume of IEEE Std 1003.1-2001 and the ISO C standard is unintentional.

This volume of IEEE Std 1003.1-2001 also allows, but does not require, mathematics functions to support IEEE Std 754-1985 and IEEE Std 854-1987.

1.8 Portability

Some of the utilities in the Shell and Utilities volume of IEEE Std 1003.1-2001 and functions in the System Interfaces volume of IEEE Std 1003.1-2001 describe functionality that might not be fully portable to systems meeting the requirements for POSIX conformance (see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 2, Conformance).

Where optional, enhanced, or reduced functionality is specified, the text is shaded and a code in the margin identifies the nature of the option, extension, or warning (see Section 1.8.1). For maximum portability, an application should avoid such functionality.

1.8.1 Codes

Margin codes and their meanings are listed in the Base Definitions volume of IEEE Std 1003.1-2001, but are repeated here for convenience:

ADV Advisory Information

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the ADV margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the ADV margin legend.

AIO Asynchronous Input and Output

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the AIO margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the AIO margin legend.

BAR Barriers

The functionality described is optional. The functionality described is also an extension to the

107		ISO C standard.
108		Where applicable, functions are marked with the BAR margin legend in the SYNOPSIS section.
109		Where additional semantics apply to a function, the material is identified by use of the BAR
110		margin legend.
111	BE	Batch Environment Services and Utilities
112		The functionality described is optional.
113		Where applicable, utilities are marked with the BE margin legend in the SYNOPSIS section.
114		Where additional semantics apply to a utility, the material is identified by use of the BE margin
115		legend.
116	CD	C-Language Development Utilities
117		The functionality described is optional.
118		Where applicable, utilities are marked with the CD margin legend in the SYNOPSIS section.
119		Where additional semantics apply to a utility, the material is identified by use of the CD margin
120		legend.
121	CPT	Process CPU-Time Clocks
122		The functionality described is optional. The functionality described is also an extension to the
123		ISO C standard.
124		Where applicable, functions are marked with the CPT margin legend in the SYNOPSIS section.
125		Where additional semantics apply to a function, the material is identified by use of the CPT
126		margin legend.
127	CS	Clock Selection
128		The functionality described is optional. The functionality described is also an extension to the
129		ISO C standard.
130		Where applicable, functions are marked with the CS margin legend in the SYNOPSIS section.
131		Where additional semantics apply to a function, the material is identified by use of the CS
132		margin legend.
133	CX	Extension to the ISO C standard
134		The functionality described is an extension to the ISO C standard. Application writers may make
135		use of an extension as it is supported on all IEEE Std 1003.1-2001-conforming systems.
136		With each function or header from the ISO C standard, a statement to the effect that “any
137		conflict is unintentional” is included. That is intended to refer to a direct conflict.
138		IEEE Std 1003.1-2001 acts in part as a profile of the ISO C standard, and it may choose to further
139		constrain behaviors allowed to vary by the ISO C standard. Such limitations are not considered
140		conflicts.
141		Where additional semantics apply to a function or header, the material is identified by use of the
142		CX margin legend.
143	FD	FORTTRAN Development Utilities
144		The functionality described is optional.
145		Where applicable, utilities are marked with the FD margin legend in the SYNOPSIS section.
146		Where additional semantics apply to a utility, the material is identified by use of the FD margin
147		legend.
148	FR	FORTTRAN Runtime Utilities
149		The functionality described is optional.

150		Where applicable, utilities are marked with the FR margin legend in the SYNOPSIS section.
151		Where additional semantics apply to a utility, the material is identified by use of the FR margin
152		legend.
153	FSC	File Synchronization
154		The functionality described is optional. The functionality described is also an extension to the
155		ISO C standard.
156		Where applicable, functions are marked with the FSC margin legend in the SYNOPSIS section.
157		Where additional semantics apply to a function, the material is identified by use of the FSC
158		margin legend.
159	IP6	IPV6
160		The functionality described is optional. The functionality described is also an extension to the
161		ISO C standard.
162		Where applicable, functions are marked with the IP6 margin legend in the SYNOPSIS section.
163		Where additional semantics apply to a function, the material is identified by use of the IP6
164		margin legend.
165	MC1	Advisory Information and either Memory Mapped Files or Shared Memory Objects
166		The functionality described is optional. The functionality described is also an extension to the
167		ISO C standard.
168		This is a shorthand notation for combinations of multiple option codes.
169		Where applicable, functions are marked with the MC1 margin legend in the SYNOPSIS section.
170		Where additional semantics apply to a function, the material is identified by use of the MC1
171		margin legend.
172		Refer to the Base Definitions volume of IEEE Std 1003.1-2001, Section 1.5.2, Margin Code
173		Notation.
174	MC2	Memory Mapped Files, Shared Memory Objects, or Memory Protection
175		The functionality described is optional. The functionality described is also an extension to the
176		ISO C standard.
177		This is a shorthand notation for combinations of multiple option codes.
178		Where applicable, functions are marked with the MC2 margin legend in the SYNOPSIS section.
179		Where additional semantics apply to a function, the material is identified by use of the MC2
180		margin legend.
181		Refer to the Base Definitions volume of IEEE Std 1003.1-2001, Section 1.5.2, Margin Code
182		Notation.
183	MF	Memory Mapped Files
184		The functionality described is optional. The functionality described is also an extension to the
185		ISO C standard.
186		Where applicable, functions are marked with the MF margin legend in the SYNOPSIS section.
187		Where additional semantics apply to a function, the material is identified by use of the MF
188		margin legend.
189	ML	Process Memory Locking
190		The functionality described is optional. The functionality described is also an extension to the
191		ISO C standard.
192		Where applicable, functions are marked with the ML margin legend in the SYNOPSIS section.
193		Where additional semantics apply to a function, the material is identified by use of the ML

194		margin legend.
195	MLR	Range Memory Locking
196		The functionality described is optional. The functionality described is also an extension to the
197		ISO C standard.
198		Where applicable, functions are marked with the MLR margin legend in the SYNOPSIS section.
199		Where additional semantics apply to a function, the material is identified by use of the MLR
200		margin legend.
201	MON	Monotonic Clock
202		The functionality described is optional. The functionality described is also an extension to the
203		ISO C standard.
204		Where applicable, functions are marked with the MON margin legend in the SYNOPSIS section.
205		Where additional semantics apply to a function, the material is identified by use of the MON
206		margin legend.
207	MPR	Memory Protection
208		The functionality described is optional. The functionality described is also an extension to the
209		ISO C standard.
210		Where applicable, functions are marked with the MPR margin legend in the SYNOPSIS section.
211		Where additional semantics apply to a function, the material is identified by use of the MPR
212		margin legend.
213	MSG	Message Passing
214		The functionality described is optional. The functionality described is also an extension to the
215		ISO C standard.
216		Where applicable, functions are marked with the MSG margin legend in the SYNOPSIS section.
217		Where additional semantics apply to a function, the material is identified by use of the MSG
218		margin legend.
219	MX	IEC 60559 Floating-Point Option
220		The functionality described is optional. The functionality described is also an extension to the
221		ISO C standard.
222		Where applicable, functions are marked with the MX margin legend in the SYNOPSIS section.
223		Where additional semantics apply to a function, the material is identified by use of the MX
224		margin legend.
225	OB	Obsolescent
226		The functionality described may be withdrawn in a future version of this volume of
227		IEEE Std 1003.1-2001. Strictly Conforming POSIX Applications and Strictly Conforming XSI
228		Applications shall not use obsolescent features.
229		Where applicable, the material is identified by use of the OB margin legend.
230	OF	Output Format Incompletely Specified
231		The functionality described is an XSI extension. The format of the output produced by the utility
232		is not fully specified. It is therefore not possible to post-process this output in a consistent
233		fashion. Typical problems include unknown length of strings and unspecified field delimiters.
234		Where applicable, the material is identified by use of the OF margin legend.
235	OH	Optional Header
236		In the SYNOPSIS section of some interfaces in the System Interfaces volume of
237		IEEE Std 1003.1-2001 an included header is marked as in the following example:

238	OH	<code>#include <sys/types.h></code>
239		<code>#include <grp.h></code>
240		<code>struct group *getgrnam(const char *name);</code>
241		The OH margin legend indicates that the marked header is not required on XSI-conformant systems.
242		
243	PIO	Prioritized Input and Output
244		The functionality described is optional. The functionality described is also an extension to the ISO C standard.
245		
246		Where applicable, functions are marked with the PIO margin legend in the SYNOPSIS section.
247		Where additional semantics apply to a function, the material is identified by use of the PIO margin legend.
248		
249	PS	Process Scheduling
250		The functionality described is optional. The functionality described is also an extension to the ISO C standard.
251		
252		Where applicable, functions are marked with the PS margin legend in the SYNOPSIS section.
253		Where additional semantics apply to a function, the material is identified by use of the PS margin legend.
254		
255	RS	Raw Sockets
256		The functionality described is optional. The functionality described is also an extension to the ISO C standard.
257		
258		Where applicable, functions are marked with the RS margin legend in the SYNOPSIS section.
259		Where additional semantics apply to a function, the material is identified by use of the RS margin legend.
260		
261	RTS	Realtime Signals Extension
262		The functionality described is optional. The functionality described is also an extension to the ISO C standard.
263		
264		Where applicable, functions are marked with the RTS margin legend in the SYNOPSIS section.
265		Where additional semantics apply to a function, the material is identified by use of the RTS margin legend.
266		
267	SD	Software Development Utilities
268		The functionality described is optional.
269		Where applicable, utilities are marked with the SD margin legend in the SYNOPSIS section.
270		Where additional semantics apply to a utility, the material is identified by use of the SD margin legend.
271		
272	SEM	Semaphores
273		The functionality described is optional. The functionality described is also an extension to the ISO C standard.
274		
275		Where applicable, functions are marked with the SEM margin legend in the SYNOPSIS section.
276		Where additional semantics apply to a function, the material is identified by use of the SEM margin legend.
277		
278	SHM	Shared Memory Objects
279		The functionality described is optional. The functionality described is also an extension to the ISO C standard.
280		
281		Where applicable, functions are marked with the SHM margin legend in the SYNOPSIS section.
282		Where additional semantics apply to a function, the material is identified by use of the SHM

283		margin legend.
284	SIO	Synchronized Input and Output
285		The functionality described is optional. The functionality described is also an extension to the
286		ISO C standard.
287		Where applicable, functions are marked with the SIO margin legend in the SYNOPSIS section.
288		Where additional semantics apply to a function, the material is identified by use of the SIO
289		margin legend.
290	SPI	Spin Locks
291		The functionality described is optional. The functionality described is also an extension to the
292		ISO C standard.
293		Where applicable, functions are marked with the SPI margin legend in the SYNOPSIS section.
294		Where additional semantics apply to a function, the material is identified by use of the SPI
295		margin legend.
296	SPN	Spawn
297		The functionality described is optional. The functionality described is also an extension to the
298		ISO C standard.
299		Where applicable, functions are marked with the SPN margin legend in the SYNOPSIS section.
300		Where additional semantics apply to a function, the material is identified by use of the SPN
301		margin legend.
302	SS	Process Sporadic Server
303		The functionality described is optional. The functionality described is also an extension to the
304		ISO C standard.
305		Where applicable, functions are marked with the SS margin legend in the SYNOPSIS section.
306		Where additional semantics apply to a function, the material is identified by use of the SS
307		margin legend.
308	TCT	Thread CPU-Time Clocks
309		The functionality described is optional. The functionality described is also an extension to the
310		ISO C standard.
311		Where applicable, functions are marked with the TCT margin legend in the SYNOPSIS section.
312		Where additional semantics apply to a function, the material is identified by use of the TCT
313		margin legend.
314	TEF	Trace Event Filter
315		The functionality described is optional. The functionality described is also an extension to the
316		ISO C standard.
317		Where applicable, functions are marked with the TEF margin legend in the SYNOPSIS section.
318		Where additional semantics apply to a function, the material is identified by use of the TEF
319		margin legend.
320	THR	Threads
321		The functionality described is optional. The functionality described is also an extension to the
322		ISO C standard.
323		Where applicable, functions are marked with the THR margin legend in the SYNOPSIS section.
324		Where additional semantics apply to a function, the material is identified by use of the THR
325		margin legend.
326	TMO	Timeouts
327		The functionality described is optional. The functionality described is also an extension to the

328		ISO C standard.
329		Where applicable, functions are marked with the TMO margin legend in the SYNOPSIS section.
330		Where additional semantics apply to a function, the material is identified by use of the TMO
331		margin legend.
332	TMR	Timers
333		The functionality described is optional. The functionality described is also an extension to the
334		ISO C standard.
335		Where applicable, functions are marked with the TMR margin legend in the SYNOPSIS section.
336		Where additional semantics apply to a function, the material is identified by use of the TMR
337		margin legend.
338	TPI	Thread Priority Inheritance
339		The functionality described is optional. The functionality described is also an extension to the
340		ISO C standard.
341		Where applicable, functions are marked with the TPI margin legend in the SYNOPSIS section.
342		Where additional semantics apply to a function, the material is identified by use of the TPI
343		margin legend.
344	TPP	Thread Priority Protection
345		The functionality described is optional. The functionality described is also an extension to the
346		ISO C standard.
347		Where applicable, functions are marked with the TPP margin legend in the SYNOPSIS section.
348		Where additional semantics apply to a function, the material is identified by use of the TPP
349		margin legend.
350	TPS	Thread Execution Scheduling
351		The functionality described is optional. The functionality described is also an extension to the
352		ISO C standard.
353		Where applicable, functions are marked with the TPS margin legend for the SYNOPSIS section.
354		Where additional semantics apply to a function, the material is identified by use of the TPS
355		margin legend.
356	TRC	Trace
357		The functionality described is optional. The functionality described is also an extension to the
358		ISO C standard.
359		Where applicable, functions are marked with the TRC margin legend in the SYNOPSIS section.
360		Where additional semantics apply to a function, the material is identified by use of the TRC
361		margin legend.
362	TRI	Trace Inherit
363		The functionality described is optional. The functionality described is also an extension to the
364		ISO C standard.
365		Where applicable, functions are marked with the TRI margin legend in the SYNOPSIS section.
366		Where additional semantics apply to a function, the material is identified by use of the TRI
367		margin legend.
368	TRL	Trace Log
369		The functionality described is optional. The functionality described is also an extension to the
370		ISO C standard.
371		Where applicable, functions are marked with the TRL margin legend in the SYNOPSIS section.
372		Where additional semantics apply to a function, the material is identified by use of the TRL

373		margin legend.
374	TSA	Thread Stack Address Attribute
375		The functionality described is optional. The functionality described is also an extension to the
376		ISO C standard.
377		Where applicable, functions are marked with the TSA margin legend for the SYNOPSIS section.
378		Where additional semantics apply to a function, the material is identified by use of the TSA
379		margin legend.
380	TSF	Thread-Safe Functions
381		The functionality described is optional. The functionality described is also an extension to the
382		ISO C standard.
383		Where applicable, functions are marked with the TSF margin legend in the SYNOPSIS section.
384		Where additional semantics apply to a function, the material is identified by use of the TSF
385		margin legend.
386	TSH	Thread Process-Shared Synchronization
387		The functionality described is optional. The functionality described is also an extension to the
388		ISO C standard.
389		Where applicable, functions are marked with the TSH margin legend in the SYNOPSIS section.
390		Where additional semantics apply to a function, the material is identified by use of the TSH
391		margin legend.
392	TSP	Thread Sporadic Server
393		The functionality described is optional. The functionality described is also an extension to the
394		ISO C standard.
395		Where applicable, functions are marked with the TSP margin legend in the SYNOPSIS section.
396		Where additional semantics apply to a function, the material is identified by use of the TSP
397		margin legend.
398	TSS	Thread Stack Address Size
399		The functionality described is optional. The functionality described is also an extension to the
400		ISO C standard.
401		Where applicable, functions are marked with the TSS margin legend in the SYNOPSIS section.
402		Where additional semantics apply to a function, the material is identified by use of the TSS
403		margin legend.
404	TYM	Typed Memory Objects
405		The functionality described is optional. The functionality described is also an extension to the
406		ISO C standard.
407		Where applicable, functions are marked with the TYM margin legend in the SYNOPSIS section.
408		Where additional semantics apply to a function, the material is identified by use of the TYM
409		margin legend.
410	UP	User Portability Utilities
411		The functionality described is optional.
412		Where applicable, utilities are marked with the UP margin legend in the SYNOPSIS section.
413		Where additional semantics apply to a utility, the material is identified by use of the UP margin
414		legend.
415	XSI	Extension
416		The functionality described is an XSI extension. Functionality marked XSI is also an extension to
417		the ISO C standard. Application writers may confidently make use of an extension on all

systems supporting the X/Open System Interfaces Extension.

If an entire SYNOPSIS section is shaded and marked XSI, all the functionality described in that reference page is an extension. See the Base Definitions volume of IEEE Std 1003.1-2001, Section 3.439, XSI.

XSR XSI STREAMS

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the XSR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the XSR margin legend.

1.9 Format of Entries

The entries in Chapter 3 are based on a common format as follows. The only sections relating to conformance are the SYNOPSIS, DESCRIPTION, RETURN VALUE, and ERRORS sections.

NAME

This section gives the name or names of the entry and briefly states its purpose.

SYNOPSIS

This section summarizes the use of the entry being described. If it is necessary to include a header to use this function, the names of such headers are shown, for example:

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

DESCRIPTION

This section describes the functionality of the function or header.

RETURN VALUE

This section indicates the possible return values, if any.

If the implementation can detect errors, “successful completion” means that no error has been detected during execution of the function. If the implementation does detect an error, the error is indicated.

For functions where no errors are defined, “successful completion” means that if the implementation checks for errors, no error has been detected. If the implementation can detect errors, and an error is detected, the indicated return value is returned and *errno* may be set.

ERRORS

This section gives the symbolic names of the error values returned by a function or stored into a variable accessed through the symbol *errno* if an error occurs.

“No errors are defined” means that error values returned by a function or stored into a variable accessed through the symbol *errno*, if any, depend on the implementation.

EXAMPLES

This section is informative.

This section gives examples of usage, where appropriate. In the event of conflict between an example and a normative part of this volume of IEEE Std 1003.1-2001, the normative material is to be taken as correct.

APPLICATION USAGE

This section is informative.

This section gives warnings and advice to application writers about the entry. In the event of conflict between warnings and advice and a normative part of this volume of IEEE Std 1003.1-2001, the normative material is to be taken as correct.

RATIONALE

This section is informative.

This section contains historical information concerning the contents of this volume of IEEE Std 1003.1-2001 and why features were included or discarded by the standard developers.

FUTURE DIRECTIONS

This section is informative.

This section provides comments which should be used as a guide to current thinking; there is not necessarily a commitment to adopt these future directions.

SEE ALSO

This section is informative.

This section gives references to related information.

CHANGE HISTORY

This section is informative.

This section shows the derivation of the entry and any significant changes that have been made to it.

General Information

This chapter covers information that is relevant to all the functions specified in Chapter 3 and the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 13, Headers.

2.1 Use and Implementation of Functions

Each of the following statements shall apply unless explicitly stated otherwise in the detailed descriptions that follow:

1. If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer outside the address space of the program, or a null pointer), the behavior is undefined.
2. Any function declared in a header may also be implemented as a macro defined in the header, so a function should not be declared explicitly if its header is included. Any macro definition of a function can be suppressed locally by enclosing the name of the function in parentheses, because the name is then not followed by the left parenthesis that indicates expansion of a macro function name. For the same syntactic reason, it is permitted to take the address of a function even if it is also defined as a macro. The use of the C-language **#undef** construct to remove any such macro definition shall also ensure that an actual function is referred to.
3. Any invocation of a function that is implemented as a macro shall expand to code that evaluates each of its arguments exactly once, fully protected by parentheses where necessary, so it is generally safe to use arbitrary expressions as arguments. Likewise, those function-like macros described in the following sections may be invoked in an expression anywhere a function with a compatible return type could be called.
4. Provided that a function can be declared without reference to any type defined in a header, it is also permissible to declare the function explicitly and use it without including its associated header.
5. If a function that accepts a variable number of arguments is not declared (explicitly or by including its associated header), the behavior is undefined.

2.2 The Compilation Environment

2.2.1 POSIX.1 Symbols

Certain symbols in this volume of IEEE Std 1003.1-2001 are defined in headers (see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 13, Headers). Some of those headers could also define symbols other than those defined by IEEE Std 1003.1-2001, potentially conflicting with symbols used by the application. Also, IEEE Std 1003.1-2001 defines symbols that are not permitted by other standards to appear in those headers without some control on the visibility of those symbols.

Symbols called “feature test macros” are used to control the visibility of symbols that might be included in a header. Implementations, future versions of IEEE Std 1003.1-2001, and other standards may define additional feature test macros.

In the compilation of an application that **#defines** a feature test macro specified by IEEE Std 1003.1-2001, no header defined by IEEE Std 1003.1-2001 shall be included prior to the definition of the feature test macro. This restriction also applies to any implementation-provided header in which these feature test macros are used. If the definition of the macro does not precede the **#include**, the result is undefined.

Feature test macros shall begin with the underscore character ('_').

2.2.1.1 The `_POSIX_C_SOURCE` Feature Test Macro

A POSIX-conforming application should ensure that the feature test macro `_POSIX_C_SOURCE` is defined before inclusion of any header.

When an application includes a header described by IEEE Std 1003.1-2001, and when this feature test macro is defined to have the value 200112L:

1. All symbols required by IEEE Std 1003.1-2001 to appear when the header is included shall be made visible.
2. Symbols that are explicitly permitted, but not required, by IEEE Std 1003.1-2001 to appear in that header (including those in reserved name spaces) may be made visible.
3. Additional symbols not required or explicitly permitted by IEEE Std 1003.1-2001 to be in that header shall not be made visible, except when enabled by another feature test macro.

Identifiers in IEEE Std 1003.1-2001 may only be undefined using the **#undef** directive as described in Section 2.1 (on page 13) or Section 2.2.2. These **#undef** directives shall follow all **#include** directives of any header in IEEE Std 1003.1-2001.

Note: The POSIX.1-1990 standard specified a macro called `_POSIX_SOURCE`. This has been superseded by `_POSIX_C_SOURCE`.

2.2.1.2 The `_XOPEN_SOURCE` Feature Test Macro

An XSI-conforming application should ensure that the feature test macro `_XOPEN_SOURCE` is defined with the value 600 before inclusion of any header. This is needed to enable the functionality described in Section 2.2.1.1 and in addition to enable the XSI extension.

Since this volume of IEEE Std 1003.1-2001 is aligned with the ISO C standard, and since all functionality enabled by `_POSIX_C_SOURCE` set equal to 200112L is enabled by `_XOPEN_SOURCE` set equal to 600, there should be no need to define `_POSIX_C_SOURCE` if `_XOPEN_SOURCE` is so defined. Therefore, if `_XOPEN_SOURCE` is set equal to 600 and `_POSIX_C_SOURCE` is set equal to 200112L, the behavior is the same as if only `_XOPEN_SOURCE` is defined and set equal to 600. However, should `_POSIX_C_SOURCE` be set to a value greater than 200112L, the behavior is unspecified.

2.2.2 The Name Space

All identifiers in this volume of IEEE Std 1003.1-2001, except *environ*, are defined in at least one of the headers, as shown in the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 13, Headers. When `_XOPEN_SOURCE` or `_POSIX_C_SOURCE` is defined, each header defines or declares some identifiers, potentially conflicting with identifiers used by the application. The set of identifiers visible to the application consists of precisely those identifiers from the header pages of the included headers, as well as additional identifiers reserved for the implementation. In addition, some headers may make visible identifiers from other headers as indicated on the relevant header pages.

Implementations may also add members to a structure or union without controlling the visibility of those members with a feature test macro, as long as a user-defined macro with the same name cannot interfere with the correct interpretation of the program. The identifiers reserved for use by the implementation are described below:

1. Each identifier with external linkage described in the header section is reserved for use as an identifier with external linkage if the header is included.
2. Each macro described in the header section is reserved for any use if the header is included.
3. Each identifier with file scope described in the header section is reserved for use as an identifier with file scope in the same name space if the header is included.

The prefixes `posix_`, `POSIX_`, and `_POSIX_` are reserved for use by IEEE Std 1003.1-2001 and other POSIX standards. Implementations may add symbols to the headers shown in the following table, provided the identifiers for those symbols begin with the corresponding reserved prefixes in the following table, and do not use the reserved prefixes `posix_`, `POSIX_`, or `_POSIX_`.

	Header	Prefix	Suffix	Complete Name
575				
576				
577				
578	AIO	<aio.h>	aio_, lio_, AIO_, LIO_	
579		<arpa/inet.h>	in_, inet_	
580		<ctype.h>	to[a-z], is[a-z]	
581		<dirent.h>	d_	
582		<errno.h>	E[0-9], E[A-Z]	
583		<fcntl.h>	l_	
584		<glob.h>	gl_	
585		<grp.h>	gr_	
586		<inttypes.h>		int[0-9a-z]*_t, uint[0-9a-z]*_t
587				
588		<limits.h>	_MAX, _MIN	
589		<locale.h>	LC_[A-Z]	
590	MSG	<mqueue.h>	mq_, MQ_	
591	XSI	<ndbm.h>	dbm_	
592		<netdb.h>	h_, n_, p_, s_	
593		<net/if.h>	if_	
594		<netinet/in.h>	in_, ip_, s_, sin_	
595	IP6		in6_, s6_, sin6_	
596	XSI	<poll.h>	pd_, ph_, ps_	
597		<pthread.h>	pthread_, PTHREAD_	
598		<pwd.h>	pw_	
599		<regex.h>	re_, rm_	
600	PS	<sched.h>	sched_, SCHED_	
601	SEM	<semaphore.h>	sem_, SEM_	
602		<signal.h>	sa_, uc_, SIG[A-Z], SIG_[A-Z]	
603	XSI		ss_, sv_	
604	RTS		si_, SI_, sigev_, SIGEV_, sival_	
605	XSI	<stropts.h>	bi_, ic_, l_, sl_, str_	
606		<stdint.h>		int[0-9a-z]*_t, uint[0-9a-z]*_t
607				
608		<stdlib.h>	str[a-z]	
609		<string.h>	str[a-z], mem[a-z], wcs[a-z]	
610	XSI	<sys/ipc.h>	ipc_	key, pad, seq
611	MF	<sys/mman.h>	shm_, MAP_, MCL_, MS_, PROT_	
612	XSI	<sys/msg.h>	msg	msg
613	XSI	<sys/resource.h>	rlim_, ru_	
614		<sys/select.h>	fd_, fds_, FD_	
615	XSI	<sys/sem.h>	sem	sem
616	XSI	<sys/shm.h>	shm	
617		<sys/socket.h>	ss_, sa_, if_, ifc_, ifru_, infu_, ifra_, msg_, cmsg_, l_	
618				
619		<sys/stat.h>	st_	
620	XSI	<sys/statvfs.h>	f_	
621		<sys/time.h>	fds_, it_, tv_, FD_	
622		<sys/times.h>	tms_	

	Header	Prefix	Suffix	Complete Name
626 XSI	<sys/uio.h>	iov_		UIO_MAXIOV
627	<sys/un.h>	sun_		
628	<sys/utsname.h>	uts_		
629 XSI	<sys/wait.h>	si_, W[A-Z], P_		
630	<termios.h>	c_		
631	<time.h>	tm_		
632 TMR		clock_, timer_, it_, tv_,		
633 TMR		CLOCK_, TIMER_		
634 XSI	<ucontext.h>	uc_, ss_		
635 XSI	<ulimit.h>	UL_		
636	<utime.h>	utim_		
637 XSI	<utmpx.h>	ut_	_LVL, _TIME, _PROCESS	
638				
639	<wchar.h>	wcs[a-z]		
640	<wctype.h>	is[a-z], to[a-z]		
641	<wordexp.h>	we_		
642	ANY header	POSIX_, _POSIX_, posix_	_t	

Note: The notation [A–Z] indicates any uppercase letter in the portable character set. The notation [a–z] indicates any lowercase letter in the portable character set. Commas and spaces in the lists of prefixes and complete names in the above table are not part of any prefix or complete name.

If any header in the following table is included, macros with the prefixes shown may be defined. After the last inclusion of a given header, an application may use identifiers with the corresponding prefixes for its own purpose, provided their use is preceded by a **#undef** of the corresponding macro.

	Header	Prefix
651		
652		
653 XSI	<dlfcn.h>	RTLD_
654	<fcntl.h>	F_, O_, S_
655 XSI	<fmtmsg.h>	MM_
656	<fnmatch.h>	FNM_
657 XSI	<ftw.h>	FTW_
658	<glob.h>	GLOB_
659	<inttypes.h>	PRI[a-z], SCN[a-z]
660 XSI	<ndbm.h>	DBM_
661	<net/if.h>	IF_
662	<netinet/in.h>	IMPLINK_, IN_, INADDR_, IP_, IPPORT_, IPPROTO_, SOCK_
663 IPv6		IPV6_, IN6_
664	<netinet/tcp.h>	TCP_
665 XSI	<nl_types.h>	NL_
666 XSI	<poll.h>	POLL_
667	<regex.h>	REG_
668	<signal.h>	SA_, SIG_[0-9a-z_],
669 XSI		BUS_, CLD_, FPE_, ILL_, POLL_, SEGV_, SI_, SS_, SV_, TRAP_
670	<stdint.h>	INT[0-9A-Z_]_MIN, INT[0-9A-Z_]_MAX, INT[0-9A-Z_]_C
671		UINT[0-9A-Z_]_MIN, UINT[0-9A-Z_]_MAX, UINT[0-9A-Z_]_C
672 XSI	<stropts.h>	FLUSH[A-Z], I_, M_, MUXID_R[A-Z], S_, SND[A-Z], STR_
673 XSI	<syslog.h>	LOG_
674 XSI	<sys/ipc.h>	IPC_
675 XSI	<sys/mman.h>	PROT_, MAP_, MS_
676 XSI	<sys/msg.h>	MSG[A-Z]
677 XSI	<sys/resource.h>	PRIO_, RLIM_, RLIMIT_, RUSAGE_
678 XSI	<sys/sem.h>	SEM_
679 XSI	<sys/shm.h>	SHM[A-Z], SHM_[A-Z]
680 XSI	<sys/socket.h>	AF_, CMSG_, MSG_, PF_, SCM_, SHUT_, SO_
681	<sys/stat.h>	S_
682 XSI	<sys/statvfs.h>	ST_
683 XSI	<sys/time.h>	FD_, ITIMER_
684 XSI	<sys/uio.h>	IOV_
685 XSI	<sys/wait.h>	BUS_, CLD_, FPE_, ILL_, POLL_, SEGV_, SI_, TRAP_
686	<termios.h>	V, I, O, TC, B[0-9] (See below.)
687	<wordexp.h>	WRDE_

688 **Note:** The notation [0–9] indicates any digit. The notation [A–Z] indicates any uppercase letter in the
689 portable character set. The notation [0–9a–z_] indicates any digit, any lowercase letter in the
690 portable character set, or underscore.

691 The following reserved names are used as exact matches for <termios.h>:

692 XSI	CBAUD	EXTB	VDSUSP
693	DEFECHO	FLUSHO	VLNEXT
694	ECHOCTL	LOBLK	VREPRINT
695	ECHOKE	PENDIN	VSTATUS
696	ECHOPRT	SWTCH	VWERASE
697	EXTA	VDISCARD	

The following identifiers are reserved regardless of the inclusion of headers:

1. All identifiers that begin with an underscore and either an uppercase letter or another underscore are always reserved for any use by the implementation.
2. All identifiers that begin with an underscore are always reserved for use as identifiers with file scope in both the ordinary identifier and tag name spaces.
3. All identifiers in the table below are reserved for use as identifiers with external linkage. Some of these identifiers do not appear in this volume of IEEE Std 1003.1-2001, but are reserved for future use by the ISO C standard.

_Exit	cexmp1	fabsf	lgammal	scalbln
abort	cexmp1f	fabsl	llabs	scalblnf
abs	cexmp1l	fclose	llrint	scalblnl
acos	cexp	fdim	llrintf	scalbn
acosf	cexp2	fdimf	llrintl	scalbnf
acosh	cexp2f	fdiml	llround	scalbnl
acoshf	cexp2l	feclearexcept	llroundf	scanf
acoshl	cexpf	fegetenv	llroundl	setbuf
acosl	cexpl	fegetexceptflag	localeconv	setjmp
acosl	cimag	fegetround	localtime	setlocale
asctime	cimagf	fehldexcept	log	setvbuf
asin	cimagl	feof	log10	signal
asinf	clearerr	feraiseexcept	log10f	sin
asinh	clgamma	ferror	log10l	sinf
asinhf	clgammaf	fesetenv	log1p	sinh
asinhf	clgammaf	fesetexceptflag	log1pf	sinhf
asinl	clock	fesetround	log1pl	sinhl
asinl	clog	fetestexcept	log2	sinl
atan	clog10	feupdateenv	log2f	sprintf
atan2	clog10f	fflush	log2l	sqrt
atan2f	clog10l	fgetc	logb	sqrtf
atan2l	clog1p	fgetpos	logbf	sqrtd
atanf	clog1pf	fgets	logbl	srand
atanf	clog1pl	fgetwc	logf	sscanf
atanh	clog2	fgetws	logl	str[a-z]*
atanh	clog2f	floor	longjmp	strtof
atanhf	clog2l	floorf	lrint	strtoimax
atanhl	clogf	floorl	lrintf	strtold
atanl	clogl	fma	lrintl	strtoll
atanl	conj	fmaf	lround	strtoull
atexit	conjf	fmal	lroundf	strtoumax
atof	conjl	fmax	lroundl	swprintf
atoi	copysign	fmaxf	malloc	swscanf
atol	copysignf	fmaxl	mblen	system
atoll	copysignl	fmin	mbrlen	tan
bsearch	cos	fminf	mbrtowc	tanf
cabs	cosf	fminl	mbsinit	tanh
cabsf	cosh	fmod	mbsrtowcs	tanhf
cabsl	coshf	fmodf	mbstowcs	tanhf
cacos	coshl	fmodl	mbtowc	tanl

746	cacosf	cosl	fopen	mem[a-z]*	tgamma
747	cacosh	cpow	fprintf	mktime	tgammaf
748	cacoshf	cpowf	fputc	modf	tgammal
749	cacoshl	cpowl	fputs	modff	time
750	cacosl	cproj	fputwc	modfl	tmpfile
751	calloc	cprojf	fputws	nan	tmpnam
752	carg	cprojl	fread	nanf	to[a-z]*
753	cargf	creal	free	nanl	trunc
754	cargl	crealf	freopen	nearbyint	truncf
755	casin	creall	frexp	nearbyintf	truncl
756	casinf	csin	frexpf	nearbyintl	ungetc
757	casinh	csinf	frexpl	nextafterf	ungetwc
758	casinhf	csinh	fscanf	nextafterl	va_end
759	casinhl	csinhf	fseek	nexttoward	vfprintf
760	casinl	csinhl	fsetpos	nexttowardf	vfprintf
761	catan	csinl	ftell	nexttowardl	vfwscanf
762	catanf	csqrt	fwide	perror	vfwscanf
763	catanh	csqrtf	fwprintf	pow	vprintf
764	catanhf	csqrtl	fwrite	powf	vscanf
765	catanhf	ctan	fwscanf	powl	vsprintf
766	catanhf	ctanf	getc	printf	vsscanf
767	catanhl	ctanl	getchar	putc	vswprintf
768	catanhl	ctgamma	getenv	putchar	vswscanf
769	catanl	ctgammaf	gets	puts	vwprintf
770	cbrt	ctgammal	getwc	putwc	vwscanf
771	cbrtf	ltime	getwchar	putwchar	wcrtomb
772	cbrtl	difftime	gmtime	qsort	wcs[a-z]*
773	ccos	div	hypotf	raise	wcstof
774	ccosf	erfcf	hypotl	rand	wcstoimax
775	ccosh	erfcl	ilogb	realloc	wcstold
776	ccoshf	erff	ilogbf	remainderf	wcstoll
777	ccoshl	erfl	ilogbl	remainderl	wcstoull
778	ccosl	errno	imaxabs	remove	wcstoumax
779	ceil	exit	imaxdiv	remquo	wctob
780	ceilf	exp	is[a-z]*	remquof	wctomb
781	ceilf	exp2	isblank	remquol	wctrans
782	ceill	exp2f	iswblank	rename	wctype
783	ceill	exp2l	labs	rewind	wcwidth
784	cerf	expf	ldexp	rint	wmem[a-z]*
785	cerfc	expl	ldexpf	rintf	wprintf
786	cerfcf	expm1	ldexpl	rintl	wscanf
787	cerfcl	expm1f	ldiv	round	
788	cerff	expm1l	ldiv	roundf	
789	cerfl	fabs	lgammaf	roundl	

Note: The notation [a-z] indicates any lowercase letter in the portable character set. The notation '*' indicates any combination of digits, letters in the portable character set, or underscore.

4. All functions and external identifiers defined in the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 13, Headers are reserved for use as identifiers with external linkage.

5. All the identifiers defined in this volume of IEEE Std 1003.1-2001 that have external linkage are always reserved for use as identifiers with external linkage.

No other identifiers are reserved.

Applications shall not declare or define identifiers with the same name as an identifier reserved in the same context. Since macro names are replaced whenever found, independent of scope and name space, macro names matching any of the reserved identifier names shall not be defined by an application if any associated header is included.

Except that the effect of each inclusion of `<assert.h>` depends on the definition of `NDEBUG`, headers may be included in any order, and each may be included more than once in a given scope, with no difference in effect from that of being included only once.

If used, the application shall ensure that a header is included outside of any external declaration or definition, and it shall be first included before the first reference to any type or macro it defines, or to any function or object it declares. However, if an identifier is declared or defined in more than one header, the second and subsequent associated headers may be included after the initial reference to the identifier. Prior to the inclusion of a header, the application shall not define any macros with names lexically identical to symbols defined by that header.

2.3 Error Numbers

Most functions can provide an error number. The means by which each function provides its error numbers is specified in its description.

Some functions provide the error number in a variable accessed through the symbol `errno`. The symbol `errno`, defined by including the `<errno.h>` header, expands to a modifiable lvalue of type `int`. It is unspecified whether `errno` is a macro or an identifier declared with external linkage. If a macro definition is suppressed in order to access an actual object, or a program defines an identifier with the name `errno`, the behavior is undefined.

The value of `errno` should only be examined when it is indicated to be valid by a function's return value. No function in this volume of IEEE Std 1003.1-2001 shall set `errno` to zero. For each thread of a process, the value of `errno` shall not be affected by function calls or assignments to `errno` by other threads.

Some functions return an error number directly as the function value. These functions return a value of zero to indicate success.

If more than one error occurs in processing a function call, any one of the possible errors may be returned, as the order of detection is undefined.

Implementations may support additional errors not included in this list, may generate errors included in this list under circumstances other than those described here, or may contain extensions or limitations that prevent some errors from occurring. The ERRORS section on each reference page specifies whether an error shall be returned, or whether it may be returned. Implementations shall not generate a different error number from the ones described here for error conditions described in this volume of IEEE Std 1003.1-2001, but may generate additional errors unless explicitly disallowed for a particular function.

Each implementation shall document, in the conformance document, situations in which each of the optional conditions defined in IEEE Std 1003.1-2001 is detected. The conformance document may also contain statements that one or more of the optional error conditions are not detected.

For functions under the Threads option for which `[EINTR]` is not listed as a possible error condition in this volume of IEEE Std 1003.1-2001, an implementation shall not return an error

840	code of [EINTR].
841	The following symbolic names identify the possible error numbers, in the context of the
842	functions specifically defined in this volume of IEEE Std 1003.1-2001; these general descriptions
843	are more precisely defined in the ERRORS sections of the functions that return them. Only these
844	symbolic names should be used in programs, since the actual value of the error number is
845	unspecified. All values listed in this section shall be unique integer constant expressions with
846	type int suitable for use in #if preprocessing directives, except as noted below. The values for all
847	these names shall be found in the <errno.h> header defined in the Base Definitions volume of
848	IEEE Std 1003.1-2001. The actual values are unspecified by this volume of IEEE Std 1003.1-2001.
849	[E2BIG]
850	Argument list too long. The sum of the number of bytes used by the new process image's
851	argument list and environment list is greater than the system-imposed limit of {ARG_MAX}
852	bytes.
853	or:
854	Lack of space in an output buffer.
855	or:
856	Argument is greater than the system-imposed maximum.
857	[EACCES]
858	Permission denied. An attempt was made to access a file in a way forbidden by its file
859	access permissions.
860	[EADDRINUSE]
861	Address in use. The specified address is in use.
862	[EADDRNOTAVAIL]
863	Address not available. The specified address is not available from the local system.
864	[EAFNOSUPPORT]
865	Address family not supported. The implementation does not support the specified address
866	family, or the specified address is not a valid address for the address family of the specified
867	socket.
868	[EAGAIN]
869	Resource temporarily unavailable. This is a temporary condition and later calls to the same
870	routine may complete normally.
871	[EALREADY]
872	Connection already in progress. A connection request is already in progress for the specified
873	socket.
874	[EBADF]
875	Bad file descriptor. A file descriptor argument is out of range, refers to no open file, or a
876	read (write) request is made to a file that is only open for writing (reading).
877	[EBADMSG]
878	Bad message. During a <i>read()</i> , <i>getmsg()</i> , <i>getpmsg()</i> , or <i>ioctl()</i> I_RECVFD request to a
879	STREAMS device, a message arrived at the head of the STREAM that is inappropriate for
880	the function receiving the message.
881	<i>read()</i> Message waiting to be read on a STREAM is not a data message.
882	<i>getmsg()</i> or <i>getpmsg()</i>
883	A file descriptor was received instead of a control message.

884	<i>ioctl()</i>	Control or data information was received instead of a file descriptor when
885		<code>I_RECVFD</code> was specified.
886	or:	
887		Bad Message. The implementation has detected a corrupted message.
888	[EBUSY]	
889		Resource busy. An attempt was made to make use of a system resource that is not currently
890		available, as it is being used by another process in a manner that would have conflicted with
891		the request being made by this process.
892	[ECANCELED]	
893		Operation canceled. The associated asynchronous operation was canceled before
894		completion.
895	[ECHILD]	
896		No child process. A <i>wait()</i> or <i>waitpid()</i> function was executed by a process that had no
897		existing or unwaited-for child process.
898	[ECONNABORTED]	
899		Connection aborted. The connection has been aborted.
900	[ECONNREFUSED]	
901		Connection refused. An attempt to connect to a socket was refused because there was no
902		process listening or because the queue of connection requests was full and the underlying
903		protocol does not support retransmissions.
904	[ECONNRESET]	
905		Connection reset. The connection was forcibly closed by the peer.
906	[EDEADLK]	
907		Resource deadlock would occur. An attempt was made to lock a system resource that
908		would have resulted in a deadlock situation.
909	[EDESTADDRREQ]	
910		Destination address required. No bind address was established.
911	[EDOM]	
912		Domain error. An input argument is outside the defined domain of the mathematical
913		function (defined in the ISO C standard).
914	[EDQUOT]	
915		Reserved.
916	[EEXIST]	
917		File exists. An existing file was mentioned in an inappropriate context; for example, as a
918		new link name in the <i>link()</i> function.
919	[EFAULT]	
920		Bad address. The system detected an invalid address in attempting to use an argument of a
921		call. The reliable detection of this error cannot be guaranteed, and when not detected may
922		result in the generation of a signal, indicating an address violation, which is sent to the
923		process.
924	[EFBIG]	
925		File too large. The size of a file would exceed the maximum file size of an implementation or
926		offset maximum established in the corresponding file description.

927	[EHOSTUNREACH]
928	Host is unreachable. The destination host cannot be reached (probably because the host is
929	down or a remote router cannot reach it).
930	[EIDRM]
931	Identifier removed. Returned during XSI interprocess communication if an identifier has
932	been removed from the system.
933	[EILSEQ]
934	Illegal byte sequence. A wide-character code has been detected that does not correspond to
935	a valid character, or a byte sequence does not form a valid wide-character code (defined in
936	the ISO C standard).
937	[EINPROGRESS]
938	Operation in progress. This code is used to indicate that an asynchronous operation has not
939	yet completed.
940	or:
941	O_NONBLOCK is set for the socket file descriptor and the connection cannot be
942	immediately established.
943	[EINTR]
944	Interrupted function call. An asynchronous signal was caught by the process during the
945	execution of an interruptible function. If the signal handler performs a normal return, the
946	interrupted function call may return this condition (see the Base Definitions volume of
947	IEEE Std 1003.1-2001, <signal.h>).
948	[EINVAL]
949	Invalid argument. Some invalid argument was supplied; for example, specifying an
950	undefined signal in a <i>signal()</i> function or a <i>kill()</i> function.
951	[EIO]
952	Input/output error. Some physical input or output error has occurred. This error may be
953	reported on a subsequent operation on the same file descriptor. Any other error-causing
954	operation on the same file descriptor may cause the [EIO] error indication to be lost.
955	[EISCONN]
956	Socket is connected. The specified socket is already connected.
957	[EISDIR]
958	Is a directory. An attempt was made to open a directory with write mode specified.
959	[ELOOP]
960	Symbolic link loop. A loop exists in symbolic links encountered during pathname
961	resolution. This error may also be returned if more than {SYMLOOP_MAX} symbolic links
962	are encountered during pathname resolution.
963	[EMFILE]
964	Too many open files. An attempt was made to open more than the maximum number of file
965	descriptors allowed in this process.
966	[EMLINK]
967	Too many links. An attempt was made to have the link count of a single file exceed
968	{LINK_MAX}.
969	[EMSGSIZE]
970	Message too large. A message sent on a transport provider was larger than an internal
971	message buffer or some other network limit.

972	or:
973	Inappropriate message buffer length.
974	[EMULTIHOP]
975	Reserved.
976	[ENAMETOOLONG]
977	Filename too long. The length of a pathname exceeds {PATH_MAX}, or a pathname
978	component is longer than {NAME_MAX}. This error may also occur when pathname
979	substitution, as a result of encountering a symbolic link during pathname resolution, results
980	in a pathname string the size of which exceeds {PATH_MAX}.
981	[ENETDOWN]
982	Network is down. The local network interface used to reach the destination is down.
983	[ENETRESET]
984	The connection was aborted by the network.
985	[ENETUNREACH]
986	Network unreachable. No route to the network is present.
987	[ENFILE]
988	Too many files open in system. Too many files are currently open in the system. The system
989	has reached its predefined limit for simultaneously open files and temporarily cannot accept
990	requests to open another one.
991	[ENOBUFS]
992	No buffer space available. Insufficient buffer resources were available in the system to
993	perform the socket operation.
994	XSR [ENODATA]
995	No message available. No message is available on the STREAM head read queue.
996	[ENODEV]
997	No such device. An attempt was made to apply an inappropriate function to a device; for
998	example, trying to read a write-only device such as a printer.
999	[ENOENT]
1000	No such file or directory. A component of a specified pathname does not exist, or the
1001	pathname is an empty string.
1002	[ENOEXEC]
1003	Executable file format error. A request is made to execute a file that, although it has the
1004	appropriate permissions, is not in the format required by the implementation for executable
1005	files.
1006	[ENOLCK]
1007	No locks available. A system-imposed limit on the number of simultaneous file and record
1008	locks has been reached and no more are currently available.
1009	[ENOLINK]
1010	Reserved.
1011	[ENOMEM]
1012	Not enough space. The new process image requires more memory than is allowed by the
1013	hardware or system-imposed memory management constraints.
1014	[ENOMSG]
1015	No message of the desired type. The message queue does not contain a message of the

1016		required type during XSI interprocess communication.
1017		[ENOPROTOOPT]
1018		Protocol not available. The protocol option specified to <i>setsockopt()</i> is not supported by the
1019		implementation.
1020		[ENOSPC]
1021		No space left on a device. During the <i>write()</i> function on a regular file or when extending a
1022		directory, there is no free space left on the device.
1023	XSRR	[ENOSR]
1024		No STREAM resources. Insufficient STREAMS memory resources are available to perform a
1025		STREAMS-related function. This is a temporary condition; it may be recovered from if other
1026		processes release resources.
1027	XSRR	[ENOSTR]
1028		Not a STREAM. A STREAM function was attempted on a file descriptor that was not
1029		associated with a STREAMS device.
1030		[ENOSYS]
1031		Function not implemented. An attempt was made to use a function that is not available in
1032		this implementation.
1033		[ENOTCONN]
1034		Socket not connected. The socket is not connected.
1035		[ENOTDIR]
1036		Not a directory. A component of the specified pathname exists, but it is not a directory,
1037		when a directory was expected.
1038		[ENOTEMPTY]
1039		Directory not empty. A directory other than an empty directory was supplied when an
1040		empty directory was expected.
1041		[ENOTSOCK]
1042		Not a socket. The file descriptor does not refer to a socket.
1043		[ENOTSUP]
1044		Not supported. The implementation does not support this feature of the Realtime Option
1045		Group.
1046		[ENOTTY]
1047		Inappropriate I/O control operation. A control function has been attempted for a file or
1048		special file for which the operation is inappropriate.
1049		[ENXIO]
1050		No such device or address. Input or output on a special file refers to a device that does not
1051		exist, or makes a request beyond the capabilities of the device. It may also occur when, for
1052		example, a tape drive is not on-line.
1053		[EOPNOTSUPP]
1054		Operation not supported on socket. The type of socket (address family or protocol) does not
1055		support the requested operation.
1056		[EOVERFLOW]
1057		Value too large to be stored in data type. An operation was attempted which would
1058		generate a value that is outside the range of values that can be represented in the relevant
1059		data type or that are allowed for a given data item.

1060	[EPERM]
1061	Operation not permitted. An attempt was made to perform an operation limited to
1062	processes with appropriate privileges or to the owner of a file or other resource.
1063	[EPIPE]
1064	Broken pipe. A write was attempted on a socket, pipe, or FIFO for which there is no process
1065	to read the data.
1066	[EPROTO]
1067	Protocol error. Some protocol error occurred. This error is device-specific, but is generally
1068	not related to a hardware failure.
1069	[EPROTONOSUPPORT]
1070	Protocol not supported. The protocol is not supported by the address family, or the protocol
1071	is not supported by the implementation.
1072	[EPROTOTYPE]
1073	Protocol wrong type for socket. The socket type is not supported by the protocol.
1074	[ERANGE]
1075	Result too large or too small. The result of the function is too large (overflow) or too small
1076	(underflow) to be represented in the available space (defined in the ISO C standard).
1077	[EROFS]
1078	Read-only file system. An attempt was made to modify a file or directory on a file system
1079	that is read-only.
1080	[ESPIPE]
1081	Invalid seek. An attempt was made to access the file offset associated with a pipe or FIFO.
1082	[ESRCH]
1083	No such process. No process can be found corresponding to that specified by the given
1084	process ID.
1085	[ESTALE]
1086	Reserved.
1087	XSR [ETIME]
1088	STREAM <i>ioctl()</i> timeout. The timer set for a STREAMS <i>ioctl()</i> call has expired. The cause of
1089	this error is device-specific and could indicate either a hardware or software failure, or a
1090	timeout value that is too short for the specific operation. The status of the <i>ioctl()</i> operation
1091	is unspecified.
1092	[ETIMEDOUT]
1093	Connection timed out. The connection to a remote machine has timed out. If the connection
1094	timed out during execution of the function that reported this error (as opposed to timing
1095	out prior to the function being called), it is unspecified whether the function has completed
1096	some or all of the documented behavior associated with a successful completion of the
1097	function.
1098	or:
1099	Operation timed out. The time limit associated with the operation was exceeded before the
1100	operation completed.
1101	[ETXTBSY]
1102	Text file busy. An attempt was made to execute a pure-procedure program that is currently
1103	open for writing, or an attempt has been made to open for writing a pure-procedure
1104	program that is being executed.

[EWOULDBLOCK]

Operation would block. An operation on a socket marked as non-blocking has encountered a situation such as no data available that otherwise would have caused the function to suspend execution.

A conforming implementation may assign the same values for [EWOULDBLOCK] and [EAGAIN].

[EXDEV]

Improper link. A link to a file on another file system was attempted.

2.3.1 Additional Error Numbers

Additional implementation-defined error numbers may be defined in `<errno.h>`.

2.4 Signal Concepts

2.4.1 Signal Generation and Delivery

A signal is said to be “generated” for (or sent to) a process or thread when the event that causes the signal first occurs. Examples of such events include detection of hardware faults, timer expiration, signals generated via the `sigevent` structure and terminal activity, as well as invocations of the `kill()` and `sigqueue()` functions. In some circumstances, the same event generates signals for multiple processes.

At the time of generation, a determination shall be made whether the signal has been generated for the process or for a specific thread within the process. Signals which are generated by some action attributable to a particular thread, such as a hardware fault, shall be generated for the thread that caused the signal to be generated. Signals that are generated in association with a process ID or process group ID or an asynchronous event, such as terminal activity, shall be generated for the process.

Each process has an action to be taken in response to each signal defined by the system (see Section 2.4.3 (on page 31)). A signal is said to be “delivered” to a process when the appropriate action for the process and signal is taken. A signal is said to be “accepted” by a process when the signal is selected and returned by one of the `sigwait()` functions.

During the time between the generation of a signal and its delivery or acceptance, the signal is said to be “pending”. Ordinarily, this interval cannot be detected by an application. However, a signal can be “blocked” from delivery to a thread. If the action associated with a blocked signal is anything other than to ignore the signal, and if that signal is generated for the thread, the signal shall remain pending until it is unblocked, it is accepted when it is selected and returned by a call to the `sigwait()` function, or the action associated with it is set to ignore the signal. Signals generated for the process shall be delivered to exactly one of those threads within the process which is in a call to a `sigwait()` function selecting that signal or has not blocked delivery of the signal. If there are no threads in a call to a `sigwait()` function selecting that signal, and if all threads within the process block delivery of the signal, the signal shall remain pending on the process until a thread calls a `sigwait()` function selecting that signal, a thread unblocks delivery of the signal, or the action associated with the signal is set to ignore the signal. If the action associated with a blocked signal is to ignore the signal and if that signal is generated for the process, it is unspecified whether the signal is discarded immediately upon generation or remains pending.

Each thread has a “signal mask” that defines the set of signals currently blocked from delivery to it. The signal mask for a thread shall be initialized from that of its parent or creating thread, or from the corresponding thread in the parent process if the thread was created as the result of a call to *fork()*. The *pthread_sigmask()*, *sigaction()*, *sigprocmask()*, and *sigsuspend()* functions control the manipulation of the signal mask.

The determination of which action is taken in response to a signal is made at the time the signal is delivered, allowing for any changes since the time of generation. This determination is independent of the means by which the signal was originally generated. If a subsequent occurrence of a pending signal is generated, it is implementation-defined as to whether the signal is delivered or accepted more than once in circumstances other than those in which queuing is required under the Realtime Signals Extension option. The order in which multiple, simultaneously pending signals outside the range SIGRTMIN to SIGRTMAX are delivered to or accepted by a process is unspecified.

When any stop signal (SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU) is generated for a process, any pending SIGCONT signals for that process shall be discarded. Conversely, when SIGCONT is generated for a process, all pending stop signals for that process shall be discarded. When SIGCONT is generated for a process that is stopped, the process shall be continued, even if the SIGCONT signal is blocked or ignored. If SIGCONT is blocked and not ignored, it shall remain pending until it is either unblocked or a stop signal is generated for the process.

An implementation shall document any condition not specified by this volume of IEEE Std 1003.1-2001 under which the implementation generates signals.

2.4.2 Realtime Signal Generation and Delivery

This section describes extensions to support realtime signal generation and delivery. This functionality is dependent on support of the Realtime Signals Extension option (and the rest of this section is not further shaded for this option).

Some signal-generating functions, such as high-resolution timer expiration, asynchronous I/O completion, interprocess message arrival, and the *sigqueue()* function, support the specification of an application-defined value, either explicitly as a parameter to the function or in a **sigevent** structure parameter. The **sigevent** structure is defined in <signal.h> and contains at least the following members:

Member Type	Member Name	Description
int	<i>sigev_notify</i>	Notification type.
int	<i>sigev_signo</i>	Signal number.
union sigval	<i>sigev_value</i>	Signal value.
void(*) (unsigned signal)	<i>sigev_notify_function</i>	Notification function.
(pthread_attr_t*)	<i>sigev_notify_attributes</i>	Notification attributes.

The *sigev_notify* member specifies the notification mechanism to use when an asynchronous event occurs. This volume of IEEE Std 1003.1-2001 defines the following values for the *sigev_notify* member:

SIGEV_NONE No asynchronous notification shall be delivered when the event of interest occurs.

SIGEV_SIGNAL The signal specified in *sigev_signo* shall be generated for the process when the event of interest occurs. If the implementation supports the Realtime Signals Extension option and if the SA_SIGINFO flag is set for that signal number, then the signal shall be queued to the process and the value

specified in *sigev_value* shall be the *si_value* component of the generated signal. If SA_SIGINFO is not set for that signal number, it is unspecified whether the signal is queued and what value, if any, is sent.

SIGEV_THREAD A notification function shall be called to perform notification.

An implementation may define additional notification mechanisms.

The *sigev_signo* member specifies the signal to be generated. The *sigev_value* member is the application-defined value to be passed to the signal-catching function at the time of the signal delivery or to be returned at signal acceptance as the *si_value* member of the **siginfo_t** structure.

The **signal** union is defined in <signal.h> and contains at least the following members:

Member Type	Member Name	Description
int	<i>sival_int</i>	Integer signal value.
void*	<i>sival_ptr</i>	Pointer signal value.

The *sival_int* member shall be used when the application-defined value is of type **int**; the *sival_ptr* member shall be used when the application-defined value is a pointer.

When a signal is generated by the *sigqueue()* function or any signal-generating function that supports the specification of an application-defined value, the signal shall be marked pending and, if the SA_SIGINFO flag is set for that signal, the signal shall be queued to the process along with the application-specified signal value. Multiple occurrences of signals so generated are queued in FIFO order. It is unspecified whether signals so generated are queued when the SA_SIGINFO flag is not set for that signal.

Signals generated by the *kill()* function or other events that cause signals to occur, such as detection of hardware faults, *alarm()* timer expiration, or terminal activity, and for which the implementation does not support queuing, shall have no effect on signals already queued for the same signal number.

When multiple unblocked signals, all in the range SIGRTMIN to SIGRTMAX, are pending, the behavior shall be as if the implementation delivers the pending unblocked signal with the lowest signal number within that range. No other ordering of signal delivery is specified.

If, when a pending signal is delivered, there are additional signals queued to that signal number, the signal shall remain pending. Otherwise, the pending indication shall be reset.

Multi-threaded programs can use an alternate event notification mechanism. When a notification is processed, and the *sigev_notify* member of the **sigevent** structure has the value SIGEV_THREAD, the function *sigev_notify_function* is called with parameter *sigev_value*.

The function shall be executed in an environment as if it were the *start_routine* for a newly created thread with thread attributes specified by *sigev_notify_attributes*. If *sigev_notify_attributes* is NULL, the behavior shall be as if the thread were created with the *detachstate* attribute set to PTHREAD_CREATE_DETACHED. Supplying an attributes structure with a *detachstate* attribute of PTHREAD_CREATE_JOINABLE results in undefined behavior. The signal mask of this thread is implementation-defined.

1232 **2.4.3 Signal Actions**

1233 There are three types of action that can be associated with a signal: SIG_DFL, SIG_IGN, or a
 1234 pointer to a function. Initially, all signals shall be set to SIG_DFL or SIG_IGN prior to entry of
 1235 the *main()* routine (see the *exec* functions). The actions prescribed by these values are as follows:

1236 SIG_DFL Signal-specific default action.

1237 The default actions for the signals defined in this volume of IEEE Std 1003.1-2001
 1238 RTS are specified under **<signal.h>**. If the Realtime Signals Extension option is
 1239 supported, the default actions for the realtime signals in the range SIGRTMIN to
 1240 SIGRTMAX shall be to terminate the process abnormally.

1241 If the default action is to stop the process, the execution of that process is
 1242 temporarily suspended. When a process stops, a SIGCHLD signal shall be
 1243 generated for its parent process, unless the parent process has set the
 1244 SA_NOCLDSTOP flag. While a process is stopped, any additional signals that are
 1245 sent to the process shall not be delivered until the process is continued, except
 1246 SIGKILL which always terminates the receiving process. A process that is a
 1247 member of an orphaned process group shall not be allowed to stop in response to
 1248 the SIGTSTP, SIGTTIN, or SIGTTOU signals. In cases where delivery of one of
 1249 these signals would stop such a process, the signal shall be discarded.

1250 Setting a signal action to SIG_DFL for a signal that is pending, and whose default
 1251 action is to ignore the signal (for example, SIGCHLD), shall cause the pending
 1252 signal to be discarded, whether or not it is blocked.

1253 The default action for SIGCONT is to resume execution at the point where the
 1254 RTS process was stopped, after first handling any pending unblocked signals. If the
 1255 Realtime Signals Extension option is supported, any queued values pending shall
 1256 be discarded and the resources used to queue them shall be released and returned
 1257 to the system for other use.

1258 XSI When a stopped process is continued, a SIGCHLD signal may be generated for its
 1259 parent process, unless the parent process has set the SA_NOCLDSTOP flag.

1260 SIG_IGN Ignore signal.

1261 Delivery of the signal shall have no effect on the process. The behavior of a process
 1262 RTS is undefined after it ignores a SIGFPE, SIGILL, SIGSEGV, or SIGBUS signal that
 1263 RTS was not generated by *kill()*, *sigqueue()*, or *raise()*.

1264 The system shall not allow the action for the signals SIGKILL or SIGSTOP to be set
 1265 to SIG_IGN.

1266 Setting a signal action to SIG_IGN for a signal that is pending shall cause the
 1267 pending signal to be discarded, whether or not it is blocked.

1268 If a process sets the action for the SIGCHLD signal to SIG_IGN, the behavior is
 1269 XSI unspecified, except as specified below.

1270 If the action for the SIGCHLD signal is set to SIG_IGN, child processes of the
 1271 calling processes shall not be transformed into zombie processes when they
 1272 terminate. If the calling process subsequently waits for its children, and the process
 1273 has no unwaited-for children that were transformed into zombie processes, it shall
 1274 block until all of its children terminate, and *wait()*, *waitid()*, and *waitpid()* shall fail
 1275 and set *errno* to [ECHILD].

1276 RTS If the Realtime Signals Extension option is supported, any queued values pending
 1277 shall be discarded and the resources used to queue them shall be released and
 1278 made available to queue other signals.

1279 *pointer to a function*

1280 Catch signal.

1281 On delivery of the signal, the receiving process is to execute the signal-catching
 1282 function at the specified address. After returning from the signal-catching function,
 1283 the receiving process shall resume execution at the point at which it was
 1284 interrupted.

1285 If the SA_SIGINFO flag for the signal is cleared, the signal-catching function shall
 1286 be entered as a C-language function call as follows:

1287 `void func(int signo);`

1288 XSI|RTS If the SA_SIGINFO flag for the signal is set, the signal-catching function shall be
 1289 entered as a C-language function call as follows:

1290 `void func(int signo, siginfo_t *info, void *context);`

1291 where *func* is the specified signal-catching function, *signo* is the signal number of
 1292 the signal being delivered, and *info* is a pointer to a **siginfo_t** structure defined in
 1293 **<signal.h>** containing at least the following members:

1294	Member Type	Member Name	Description
1295	int	<i>si_signo</i>	Signal number.
1296	int	<i>si_code</i>	Cause of the signal.
1297	union sigval	<i>si_value</i>	Signal value.
1298			

1299 The *si_signo* member shall contain the signal number. This shall be the same as the
 1300 *signo* parameter. The *si_code* member shall contain a code identifying the cause of
 1301 the signal. The following values are defined for *si_code*:

1302 SI_USER The signal was sent by the *kill()* function. The implementation
 1303 may set *si_code* to SI_USER if the signal was sent by the *raise()* or
 1304 *abort()* functions or any similar functions provided as
 1305 implementation extensions.

1306 RTS SI_QUEUE The signal was sent by the *sigqueue()* function.

1307 RTS SI_TIMER The signal was generated by the expiration of a timer set by
 1308 *timer_settime()*.

1309 RTS SI_ASYNCIO The signal was generated by the completion of an asynchronous
 1310 I/O request.

1311 RTS SI_MESGQ The signal was generated by the arrival of a message on an
 1312 empty message queue.

1313 If the signal was not generated by one of the functions or events listed above, the
 1314 *si_code* shall be set to an implementation-defined value that is not equal to any of
 1315 the values defined above.

1316 RTS If the Realtime Signals Extension is supported, and *si_code* is one of SI_QUEUE,
 1317 SI_TIMER, SI_ASYNCIO, or SI_MESGQ, then *si_value* shall contain the
 1318 application-specified signal value. Otherwise, the contents of *si_value* are
 1319 undefined.

1320 The behavior of a process is undefined after it returns normally from a signal-
 1321 XSI catching function for a SIGBUS, SIGFPE, SIGILL, or SIGSEGV signal that was not
 1322 RTS generated by *kill()*, *sigqueue()*, or *raise()*.

1323 The system shall not allow a process to catch the signals SIGKILL and SIGSTOP.

1324 If a process establishes a signal-catching function for the SIGCHLD signal while it
 1325 has a terminated child process for which it has not waited, it is unspecified
 1326 whether a SIGCHLD signal is generated to indicate that child process.

1327 When signal-catching functions are invoked asynchronously with process
 1328 execution, the behavior of some of the functions defined by this volume of
 1329 IEEE Std 1003.1-2001 is unspecified if they are called from a signal-catching
 1330 function.

1331 The following table defines a set of functions that shall be either reentrant or non-
 1332 interruptible by signals and shall be async-signal-safe. Therefore applications may
 1333 invoke them, without restriction, from signal-catching functions:

1334	<i>_Exit()</i>	<i>fpathconf()</i>	<i>raise()</i>	<i>sigprocmask()</i>
1335	<i>_exit()</i>	<i>fstat()</i>	<i>read()</i>	<i>sigqueue()</i>
1336	<i>accept()</i>	<i>fsync()</i>	<i>readlink()</i>	<i>sigset()</i>
1337	<i>access()</i>	<i>ftruncate()</i>	<i>recv()</i>	<i>sigsuspend()</i>
1338	<i>aio_error()</i>	<i>getgid()</i>	<i>recvfrom()</i>	<i>socket()</i>
1339	<i>aio_return()</i>	<i>geteuid()</i>	<i>recvmsg()</i>	<i>socketpair()</i>
1340	<i>aio_suspend()</i>	<i>getgid()</i>	<i>rename()</i>	<i>stat()</i>
1341	<i>alarm()</i>	<i>getgroups()</i>	<i>rmdir()</i>	<i>symlink()</i>
1342	<i>bind()</i>	<i>getpeername()</i>	<i>select()</i>	<i>sysconf()</i>
1343	<i>cfgetispeed()</i>	<i>getpgrp()</i>	<i>sem_post()</i>	<i>tcdrain()</i>
1344	<i>cfgetospeed()</i>	<i>getpid()</i>	<i>send()</i>	<i>tcflow()</i>
1345	<i>cfsetispeed()</i>	<i>getppid()</i>	<i>sendmsg()</i>	<i>tcflush()</i>
1346	<i>cfsetospeed()</i>	<i>getsockname()</i>	<i>sendto()</i>	<i>tcgetattr()</i>
1347	<i>chdir()</i>	<i>getsockopt()</i>	<i>setgid()</i>	<i>tcgetpgrp()</i>
1348	<i>chmod()</i>	<i>getuid()</i>	<i>setpgid()</i>	<i>tcsendbreak()</i>
1349	<i>chown()</i>	<i>kill()</i>	<i>setsid()</i>	<i>tcsetattr()</i>
1350	<i>clock_gettime()</i>	<i>link()</i>	<i>setsockopt()</i>	<i>tcsetpgrp()</i>
1351	<i>close()</i>	<i>listen()</i>	<i>setuid()</i>	<i>time()</i>
1352	<i>connect()</i>	<i>lseek()</i>	<i>shutdown()</i>	<i>timer_getoverrun()</i>
1353	<i>creat()</i>	<i>lstat()</i>	<i>sigaction()</i>	<i>timer_gettime()</i>
1354	<i>dup()</i>	<i>mkdir()</i>	<i>sigaddset()</i>	<i>timer_settime()</i>
1355	<i>dup2()</i>	<i>mkfifo()</i>	<i>sigdelset()</i>	<i>times()</i>
1356	<i>execle()</i>	<i>open()</i>	<i>sigemptyset()</i>	<i>umask()</i>
1357	<i>execve()</i>	<i>pathconf()</i>	<i>sigfillset()</i>	<i>uname()</i>
1358	<i>fchmod()</i>	<i>pause()</i>	<i>sigismember()</i>	<i>unlink()</i>
1359	<i>fchown()</i>	<i>pipe()</i>	<i>sleep()</i>	<i>utime()</i>
1360	<i>fcntl()</i>	<i>poll()</i>	<i>signal()</i>	<i>wait()</i>
1361	<i>fdatasync()</i>	<i>posix_trace_event()</i>	<i>sigpause()</i>	<i>waitpid()</i>
1362	<i>fork()</i>	<i>pselect()</i>	<i>sigpending()</i>	<i>write()</i>

1363 All functions not in the above table are considered to be unsafe with respect to
 1364 signals. In the presence of signals, all functions defined by this volume of
 1365 IEEE Std 1003.1-2001 shall behave as defined when called from or interrupted by a
 1366 signal-catching function, with a single exception: when a signal interrupts an
 1367 unsafe function and the signal-catching function calls an unsafe function, the

1368 behavior is undefined.

1369 When a signal is delivered to a thread, if the action of that signal specifies termination, stop, or
1370 continue, the entire process shall be terminated, stopped, or continued, respectively.

1371 **2.4.4 Signal Effects on Other Functions**

1372 Signals affect the behavior of certain functions defined by this volume of IEEE Std 1003.1-2001 if
1373 delivered to a process while it is executing such a function. If the action of the signal is to
1374 terminate the process, the process shall be terminated and the function shall not return. If the
1375 action of the signal is to stop the process, the process shall stop until continued or terminated.
1376 Generation of a SIGCONT signal for the process shall cause the process to be continued, and the
1377 original function shall continue at the point the process was stopped. If the action of the signal is
1378 to invoke a signal-catching function, the signal-catching function shall be invoked; in this case
1379 the original function is said to be “interrupted” by the signal. If the signal-catching function
1380 executes a **return** statement, the behavior of the interrupted function shall be as described
1381 individually for that function, except as noted for unsafe functions. Signals that are ignored shall
1382 not affect the behavior of any function; signals that are blocked shall not affect the behavior of
1383 any function until they are unblocked and then delivered, except as specified for the *sigpending()*
1384 and *sigwait()* functions.

1385 **2.5 Standard I/O Streams**

1386 A stream is associated with an external file (which may be a physical device) by “opening” a file,
1387 which may involve “creating” a new file. Creating an existing file causes its former contents to
1388 be discarded if necessary. If a file can support positioning requests (such as a disk file, as
1389 opposed to a terminal), then a “file position indicator” associated with the stream is positioned
1390 at the start (byte number 0) of the file, unless the file is opened with append mode, in which case
1391 it is implementation-defined whether the file position indicator is initially positioned at the
1392 beginning or end of the file. The file position indicator is maintained by subsequent reads,
1393 writes, and positioning requests, to facilitate an orderly progression through the file. All input
1394 takes place as if bytes were read by successive calls to *fgetc()*; all output takes place as if bytes
1395 were written by successive calls to *fputc()*.

1396 When a stream is “unbuffered”, bytes are intended to appear from the source or at the
1397 destination as soon as possible; otherwise, bytes may be accumulated and transmitted as a block.
1398 When a stream is “fully buffered”, bytes are intended to be transmitted as a block when a buffer
1399 is filled. When a stream is “line buffered”, bytes are intended to be transmitted as a block when a
1400 newline byte is encountered. Furthermore, bytes are intended to be transmitted as a block when
1401 a buffer is filled, when input is requested on an unbuffered stream, or when input is requested
1402 on a line-buffered stream that requires the transmission of bytes. Support for these
1403 characteristics is implementation-defined, and may be affected via *setbuf()* and *setvbuf()*.

1404 A file may be disassociated from a controlling stream by “closing” the file. Output streams are
1405 flushed (any unwritten buffer contents are transmitted) before the stream is disassociated from
1406 the file. The value of a pointer to a **FILE** object is unspecified after the associated file is closed
1407 (including the standard streams).

1408 A file may be subsequently reopened, by the same or another program execution, and its
1409 contents reclaimed or modified (if it can be repositioned at its start). If the *main()* function
1410 returns to its original caller, or if the *exit()* function is called, all open files are closed (hence all
1411 output streams are flushed) before program termination. Other paths to program termination,
1412 such as calling *abort()*, need not close all files properly.

The address of the **FILE** object used to control a stream may be significant; a copy of a **FILE** object need not necessarily serve in place of the original.

At program start-up, three streams are predefined and need not be opened explicitly: *standard input* (for reading conventional input), *standard output* (for writing conventional output), and *standard error* (for writing diagnostic output). When opened, the standard error stream is not fully buffered; the standard input and standard output streams are fully buffered if and only if the stream can be determined not to refer to an interactive device.

2.5.1 Interaction of File Descriptors and Standard I/O Streams

This section describes the interaction of file descriptors and standard I/O streams. This functionality is an extension to the ISO C standard (and the rest of this section is not further CX shaded).

An open file description may be accessed through a file descriptor, which is created using functions such as *open()* or *pipe()*, or through a stream, which is created using functions such as *fopen()* or *popen()*. Either a file descriptor or a stream is called a “handle” on the open file description to which it refers; an open file description may have several handles.

Handles can be created or destroyed by explicit user action, without affecting the underlying open file description. Some of the ways to create them include *fcntl()*, *dup()*, *fdopen()*, *fileno()*, and *fork()*. They can be destroyed by at least *fclose()*, *close()*, and the *exec* functions.

A file descriptor that is never used in an operation that could affect the file offset (for example, *read()*, *write()*, or *lseek()*) is not considered a handle for this discussion, but could give rise to one (for example, as a consequence of *fdopen()*, *dup()*, or *fork()*). This exception does not include the file descriptor underlying a stream, whether created with *fopen()* or *fdopen()*, so long as it is not used directly by the application to affect the file offset. The *read()* and *write()* functions implicitly affect the file offset; *lseek()* explicitly affects it.

The result of function calls involving any one handle (the “active handle”) is defined elsewhere in this volume of IEEE Std 1003.1-2001, but if two or more handles are used, and any one of them is a stream, the application shall ensure that their actions are coordinated as described below. If this is not done, the result is undefined.

A handle which is a stream is considered to be closed when either an *fclose()* or *freopen()* is executed on it (the result of *freopen()* is a new stream, which cannot be a handle on the same open file description as its previous value), or when the process owning that stream terminates with *exit()*, *abort()*, or due to a signal. A file descriptor is closed by *close()*, *_exit()*, or the *exec* functions when *FD_CLOEXEC* is set on that file descriptor.

For a handle to become the active handle, the application shall ensure that the actions below are performed between the last use of the handle (the current active handle) and the first use of the second handle (the future active handle). The second handle then becomes the active handle. All activity by the application affecting the file offset on the first handle shall be suspended until it again becomes the active file handle. (If a stream function has as an underlying function one that affects the file offset, the stream function shall be considered to affect the file offset.)

The handles need not be in the same process for these rules to apply.

Note that after a *fork()*, two handles exist where one existed before. The application shall ensure that, if both handles can ever be accessed, they are both in a state where the other could become the active handle first. The application shall prepare for a *fork()* exactly as if it were a change of active handle. (If the only action performed by one of the processes is one of the *exec* functions or *_exit()* (not *exit()*), the handle is never accessed in that process.)

For the first handle, the first applicable condition below applies. After the actions required below are taken, if the handle is still open, the application can close it.

- If it is a file descriptor, no action is required.
- If the only further action to be performed on any handle to this open file descriptor is to close it, no action need be taken.
- If it is a stream which is unbuffered, no action need be taken.
- If it is a stream which is line buffered, and the last byte written to the stream was a <newline> (that is, as if a:

```
putc( '\n' )
```

was the most recent operation on that stream), no action need be taken.
- If it is a stream which is open for writing or appending (but not also open for reading), the application shall either perform an *fflush()*, or the stream shall be closed.
- If the stream is open for reading and it is at the end of the file (*feof()* is true), no action need be taken.
- If the stream is open with a mode that allows reading and the underlying open file description refers to a device that is capable of seeking, the application shall either perform an *fflush()*, or the stream shall be closed.

Otherwise, the result is undefined.

For the second handle:

- If any previous active handle has been used by a function that explicitly changed the file offset, except as required above for the first handle, the application shall perform an *lseek()* or *fseek()* (as appropriate to the type of handle) to an appropriate location.

If the active handle ceases to be accessible before the requirements on the first handle, above, have been met, the state of the open file description becomes undefined. This might occur during functions such as a *fork()* or *_exit()*.

The *exec* functions make inaccessible all streams that are open at the time they are called, independent of which streams or file descriptors may be available to the new process image.

When these rules are followed, regardless of the sequence of handles used, implementations shall ensure that an application, even one consisting of several processes, shall yield correct results: no data shall be lost or duplicated when writing, and all data shall be written in order, except as requested by seeks. It is implementation-defined whether, and under what conditions, all input is seen exactly once.

If the rules above are not followed, the result is unspecified.

Each function that operates on a stream is said to have zero or more “underlying functions”. This means that the stream function shares certain traits with the underlying functions, but does not require that there be any relation between the implementations of the stream function and its underlying functions.

1495 2.5.2 Stream Orientation and Encoding Rules

1496 For conformance to the ISO/IEC 9899:1999 standard, the definition of a stream includes an
 1497 “orientation”. After a stream is associated with an external file, but before any operations are
 1498 performed on it, the stream is without orientation. Once a wide-character input/output function
 1499 has been applied to a stream without orientation, the stream shall become “wide-oriented”.
 1500 Similarly, once a byte input/output function has been applied to a stream without orientation,
 1501 the stream shall become “byte-oriented”. Only a call to the *freopen()* function or the *fwide()*
 1502 function can otherwise alter the orientation of a stream.

1503 A successful call to *freopen()* shall remove any orientation. The three predefined streams *standard*
 1504 *input*, *standard output*, and *standard error* shall be unoriented at program start-up.

1505 Byte input/output functions cannot be applied to a wide-oriented stream, and wide-character
 1506 input/output functions cannot be applied to a byte-oriented stream. The remaining stream
 1507 operations shall not affect and shall not be affected by a stream's orientation, except for the
 1508 following additional restriction:

- 1509 • For wide-oriented streams, after a successful call to a file-positioning function that leaves the
 1510 file position indicator prior to the end-of-file, a wide-character output function can overwrite
 1511 a partial character; any file contents beyond the byte(s) written are henceforth undefined.

1512 Each wide-oriented stream has an associated **mbstate_t** object that stores the current parse state
 1513 of the stream. A successful call to *fgetpos()* shall store a representation of the value of this
 1514 **mbstate_t** object as part of the value of the **fpos_t** object. A later successful call to *fsetpos()* using
 1515 the same stored **fpos_t** value shall restore the value of the associated **mbstate_t** object as well as
 1516 the position within the controlled stream.

1517 Implementations that support multiple encoding rules associate an encoding rule with the
 1518 stream. The encoding rule shall be determined by the setting of the *LC_CTYPE* category in the
 1519 current locale at the time when the stream becomes wide-oriented. As with the stream's
 1520 orientation, the encoding rule associated with a stream cannot be changed once it has been set,
 1521 except by a successful call to *freopen()* which clears the encoding rule and resets the orientation
 1522 to unoriented.

1523 Although wide-oriented streams are conceptually sequences of wide characters, the external file
 1524 associated with a wide-oriented stream is a sequence of (possibly multi-byte) characters
 1525 generalized as follows:

- 1526 • Multi-byte encodings within files may contain embedded null bytes (unlike multi-byte
 1527 encodings valid for use internal to the program).
- 1528 • A file need not begin nor end in the initial shift state.

1529 Moreover, the encodings used for characters may differ among files. Both the nature and choice
 1530 of such encodings are implementation-defined.

1531 The wide-character input functions read characters from the stream and convert them to wide
 1532 characters as if they were read by successive calls to the *fgetwc()* function. Each conversion shall
 1533 occur as if by a call to the *mbrtowc()* function, with the conversion state described by the stream's
 1534 CX own **mbstate_t** object, except the encoding rule associated with the stream is used instead of the
 1535 encoding rule implied by the *LC_CTYPE* category of the current locale.

1536 The wide-character output functions convert wide characters to (possibly multi-byte) characters
 1537 and write them to the stream as if they were written by successive calls to the *fputwc()* function.
 1538 Each conversion shall occur as if by a call to the *wcrtomb()* function, with the conversion state
 1539 CX described by the stream's own **mbstate_t** object, except the encoding rule associated with the
 1540 stream is used instead of the encoding rule implied by the *LC_CTYPE* category of the current

locale.

An “encoding error” shall occur if the character sequence presented to the underlying *mbrtowc()* function does not form a valid (generalized) character, or if the code value passed to the underlying *wcrtomb()* function does not correspond to a valid (generalized) character. The wide-character input/output functions and the byte input/output functions store the value of the macro `[EILSEQ]` in *errno* if and only if an encoding error occurs.

2.6 STREAMS

STREAMS functionality is provided on implementations supporting the XSI STREAMS Option Group. This functionality is dependent on support of the XSI STREAMS option (and the rest of this section is not further shaded for this option).

STREAMS provides a uniform mechanism for implementing networking services and other character-based I/O. The STREAMS function provides direct access to protocol modules. STREAMS modules are unspecified objects. Access to STREAMS modules is provided by interfaces in IEEE Std 1003.1-2001. Creation of STREAMS modules is outside the scope of IEEE Std 1003.1-2001.

A STREAM is typically a full-duplex connection between a process and an open device or pseudo-device. However, since pipes may be STREAMS-based, a STREAM can be a full-duplex connection between two processes. The STREAM itself exists entirely within the implementation and provides a general character I/O function for processes. It optionally includes one or more intermediate processing modules that are interposed between the process end of the STREAM (STREAM head) and a device driver at the end of the STREAM (STREAM end).

STREAMS I/O is based on messages. There are three types of message:

- *Data messages* containing actual data for input or output
- *Control data* containing instructions for the STREAMS modules and underlying implementation
- Other messages, which include file descriptors

The interface between the STREAM and the rest of the implementation is provided by a set of functions at the STREAM head. When a process calls *write()*, *writev()*, *putmsg()*, *putpmsg()*, or *ioctl()*, messages are sent down the STREAM, and *read()*, *readv()*, *getmsg()*, or *getpmsg()* accepts data from the STREAM and passes it to a process. Data intended for the device at the downstream end of the STREAM is packaged into messages and sent downstream, while data and signals from the device are composed into messages by the device driver and sent upstream to the STREAM head.

When a STREAMS-based device is opened, a STREAM shall be created that contains the STREAM head and the STREAM end (driver). If pipes are STREAMS-based in an implementation, when a pipe is created, two STREAMS shall be created, each containing a STREAM head. Other modules are added to the STREAM using *ioctl()*. New modules are “pushed” onto the STREAM one at a time in last-in, first-out (LIFO) style, as though the STREAM was a push-down stack.

Priority

Message types are classified according to their queuing priority and may be *normal* (non-priority), *priority*, or *high-priority* messages. A message belongs to a particular priority band that determines its ordering when placed on a queue. Normal messages have a priority band of 0 and shall always be placed at the end of the queue following all other messages in the queue. High-priority messages are always placed at the head of a queue, but shall be discarded if there is already a high-priority message in the queue. Their priority band shall be ignored; they are high-priority by virtue of their type. Priority messages have a priority band greater than 0. Priority messages are always placed after any messages of the same or higher priority. High-priority and priority messages are used to send control and data information outside the normal flow of control. By convention, high-priority messages shall not be affected by flow control. Normal and priority messages have separate flow controls.

Message Parts

A process may access STREAMS messages that contain a data part, control part, or both. The data part is that information which is transmitted over the communication medium and the control information is used by the local STREAMS modules. The other types of messages are used between modules and are not accessible to processes. Messages containing only a data part are accessible via *putmsg()*, *putpmsg()*, *getmsg()*, *getpmsg()*, *read()*, *readv()*, *write()*, or *writv()*. Messages containing a control part with or without a data part are accessible via calls to *putmsg()*, *putpmsg()*, *getmsg()*, or *getpmsg()*.

2.6.1 Accessing STREAMS

A process accesses STREAMS-based files using the standard functions *close()*, *ioctl()*, *getmsg()*, *getpmsg()*, *open()*, *pipe()*, *poll()*, *putmsg()*, *putpmsg()*, *read()*, or *write()*. Refer to the applicable function definitions for general properties and errors.

Calls to *ioctl()* shall perform control functions on the STREAM associated with the file descriptor *fildev*. The control functions may be performed by the STREAM head, a STREAMS module, or the STREAMS driver for the STREAM.

STREAMS modules and drivers can detect errors, sending an error message to the STREAM head, thus causing subsequent functions to fail and set *errno* to the value specified in the message. In addition, STREAMS modules and drivers can elect to fail a particular *ioctl()* request alone by sending a negative acknowledgement message to the STREAM head. This shall cause just the pending *ioctl()* request to fail and set *errno* to the value specified in the message.

2.7 XSI Interprocess Communication

XSI

This section describes extensions to support interprocess communication. This functionality is dependent on support of the XSI extension (and the rest of this section is not further shaded for this option).

The following message passing, semaphore, and shared memory services form an XSI interprocess communication facility. Certain aspects of their operation are common, and are defined as follows.

IPC Functions		
<i>msgctl()</i>	<i>semctl()</i>	<i>shmctl()</i>
<i>msgget()</i>	<i>semget()</i>	<i>shmdt()</i>
<i>msgrcv()</i>	<i>semop()</i>	<i>shmget()</i>
<i>msgsnd()</i>	<i>shmat()</i>	

Another interprocess communication facility is provided by functions in the Realtime Option Group; see Section 2.8 (on page 41).

2.7.1 IPC General Description

Each individual shared memory segment, message queue, and semaphore set shall be identified by a unique positive integer, called, respectively, a shared memory identifier, *shmid*, a semaphore identifier, *semid*, and a message queue identifier, *msqid*. The identifiers shall be returned by calls to *shmget()*, *semget()*, and *msgget()*, respectively.

Associated with each identifier is a data structure which contains data related to the operations which may be or may have been performed; see the Base Definitions volume of IEEE Std 1003.1-2001, <sys/shm.h>, <sys/sem.h>, and <sys/msg.h> for their descriptions.

Each of the data structures contains both ownership information and an **ipc_perm** structure (see the Base Definitions volume of IEEE Std 1003.1-2001, <sys/ipc.h>) which are used in conjunction to determine whether or not read/write (read/alter for semaphores) permissions should be granted to processes using the IPC facilities. The *mode* member of the **ipc_perm** structure acts as a bit field which determines the permissions.

The values of the bits are given below in octal notation.

Bit	Meaning
0400	Read by user.
0200	Write by user.
0040	Read by group.
0020	Write by group.
0004	Read by others.
0002	Write by others.

The name of the **ipc_perm** structure is *shm_perm*, *sem_perm*, or *msg_perm*, depending on which service is being used. In each case, read and write/alter permissions shall be granted to a process if one or more of the following are true ("xxx" is replaced by *shm*, *sem*, or *msg*, as appropriate):

- The process has appropriate privileges.
- The effective user ID of the process matches *xxx_perm.cuid* or *xxx_perm.uid* in the data structure associated with the IPC identifier, and the appropriate bit of the *user* field in *xxx_perm.mode* is set.
- The effective user ID of the process does not match *xxx_perm.cuid* or *xxx_perm.uid* but the effective group ID of the process matches *xxx_perm.cgid* or *xxx_perm.gid* in the data structure associated with the IPC identifier, and the appropriate bit of the *group* field in *xxx_perm.mode* is set.
- The effective user ID of the process does not match *xxx_perm.cuid* or *xxx_perm.uid* and the effective group ID of the process does not match *xxx_perm.cgid* or *xxx_perm.gid* in the data structure associated with the IPC identifier, but the appropriate bit of the *other* field in *xxx_perm.mode* is set.

1664 Otherwise, the permission shall be denied.

1665 2.8 Realtime

1666 This section defines functions to support the source portability of applications with realtime
1667 requirements. The presence of many of these functions is dependent on support for
1668 implementation options described in the text.

1669 The specific functional areas included in this section and their scope include the following. Full
1670 definitions of these terms can be found in the Base Definitions volume of IEEE Std 1003.1-2001,
1671 Chapter 3, Definitions.

- 1672 • Semaphores
- 1673 • Process Memory Locking
- 1674 • Memory Mapped Files and Shared Memory Objects
- 1675 • Priority Scheduling
- 1676 • Realtime Signal Extension
- 1677 • Timers
- 1678 • Interprocess Communication
- 1679 • Synchronized Input and Output
- 1680 • Asynchronous Input and Output

1681 All the realtime functions defined in this volume of IEEE Std 1003.1-2001 are portable, although
1682 some of the numeric parameters used by an implementation may have hardware dependencies.

1683 2.8.1 Realtime Signals

1684 RTS Realtime signal generation and delivery is dependent on support for the Realtime Signals
1685 Extension option.

1686 See Section 2.4.2 (on page 29).

1687 2.8.2 Asynchronous I/O

1688 AIO The functionality described in this section is dependent on support of the Asynchronous Input
1689 and Output option (and the rest of this section is not further shaded for this option).

1690 An asynchronous I/O control block structure **aio_cblock** is used in many asynchronous I/O
1691 functions. It is defined in the Base Definitions volume of IEEE Std 1003.1-2001, **<aio.h>** and has
1692 at least the following members:

1693	Member Type	Member Name	Description
1694	int	<i>aio_fildes</i>	File descriptor.
1695	off_t	<i>aio_offset</i>	File offset.
1696	volatile void*	<i>aio_buf</i>	Location of buffer.
1697	size_t	<i>aio_nbytes</i>	Length of transfer.
1698	int	<i>aio_reqprio</i>	Request priority offset.
1699	struct sigevent	<i>aio_sigevent</i>	Signal number and value.
1700	int	<i>aio_lio_opcode</i>	Operation to be performed.

The *aio_fildes* element is the file descriptor on which the asynchronous operation is performed.

If `O_APPEND` is not set for the file descriptor *aio_fildes* and if *aio_fildes* is associated with a device that is capable of seeking, then the requested operation takes place at the absolute position in the file as given by *aio_offset*, as if *lseek()* were called immediately prior to the operation with an *offset* argument equal to *aio_offset* and a *whence* argument equal to `SEEK_SET`. If `O_APPEND` is set for the file descriptor, or if *aio_fildes* is associated with a device that is incapable of seeking, write operations append to the file in the same order as the calls were made, with the following exception: under implementation-defined circumstances, such as operation on a multi-processor or when requests of differing priorities are submitted at the same time, the ordering restriction may be relaxed. Since there is no way for a strictly conforming application to determine whether this relaxation applies, all strictly conforming applications which rely on ordering of output shall be written in such a way that they will operate correctly if the relaxation applies. After a successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file is unspecified. The *aio_nbytes* and *aio_buf* elements are the same as the *nbyte* and *buf* arguments defined by *read()* and *write()*, respectively.

If `_POSIX_PRIORITIZED_IO` and `_POSIX_PRIORITY_SCHEDULING` are defined, then asynchronous I/O is queued in priority order, with the priority of each asynchronous operation based on the current scheduling priority of the calling process. The *aio_reqprio* member can be used to lower (but not raise) the asynchronous I/O operation priority and is within the range zero through `{AIO_PRIO_DELTA_MAX}`, inclusive. Unless both `_POSIX_PRIORITIZED_IO` and `_POSIX_PRIORITY_SCHEDULING` are defined, the order of processing asynchronous I/O requests is unspecified. When both `_POSIX_PRIORITIZED_IO` and `_POSIX_PRIORITY_SCHEDULING` are defined, the order of processing of requests submitted by processes whose schedulers are not `SCHED_FIFO`, `SCHED_RR`, or `SCHED_SPORADIC` is unspecified. The priority of an asynchronous request is computed as (process scheduling priority) minus *aio_reqprio*. The priority assigned to each asynchronous I/O request is an indication of the desired order of execution of the request relative to other asynchronous I/O requests for this file. If `_POSIX_PRIORITIZED_IO` is defined, requests issued with the same priority to a character special file are processed by the underlying device in FIFO order; the order of processing of requests of the same priority issued to files that are not character special files is unspecified. Numerically higher priority values indicate requests of higher priority. The value of *aio_reqprio* has no effect on process scheduling priority. When prioritized asynchronous I/O requests to the same file are blocked waiting for a resource required for that I/O operation, the higher-priority I/O requests shall be granted the resource before lower-priority I/O requests are granted the resource. The relative priority of asynchronous I/O and synchronous I/O is implementation-defined. If `_POSIX_PRIORITIZED_IO` is defined, the implementation shall define for which files I/O prioritization is supported.

The *aio_sigevent* determines how the calling process shall be notified upon I/O completion, as specified in Section 2.4.1 (on page 28). If *aio_sigevent.sigev_notify* is `SIGEV_NONE`, then no signal shall be posted upon I/O completion, but the error status for the operation and the return status for the operation shall be set appropriately.

The *aio_lio_opcode* field is used only by the *lio_listio()* call. The *lio_listio()* call allows multiple asynchronous I/O operations to be submitted at a single time. The function takes as an argument an array of pointers to **aiocb** structures. Each **aiocb** structure indicates the operation to be performed (read or write) via the *aio_lio_opcode* field.

The address of the **aiocb** structure is used as a handle for retrieving the error status and return status of the asynchronous operation while it is in progress.

The **aiocb** structure and the data buffers associated with the asynchronous I/O operation are being used by the system for asynchronous I/O while, and only while, the error status of the

1750 asynchronous operation is equal to [EINPROGRESS]. Applications shall not modify the **aiocb**
 1751 structure while the structure is being used by the system for asynchronous I/O.

1752 The return status of the asynchronous operation is the number of bytes transferred by the I/O
 1753 operation. If the error status is set to indicate an error completion, then the return status is set to
 1754 the return value that the corresponding *read()*, *write()*, or *fsync()* call would have returned.
 1755 When the error status is not equal to [EINPROGRESS], the return status shall reflect the return
 1756 status of the corresponding synchronous operation.

1757 2.8.3 Memory Management

1758 2.8.3.1 Memory Locking

1759 ML The functionality described in this section is dependent on support of the Process Memory
 1760 Locking option (and the rest of this section is not further shaded for this option).

1761 Range memory locking operations are defined in terms of pages. Implementations may restrict
 1762 the size and alignment of range lockings to be on page-size boundaries. The page size, in bytes,
 1763 is the value of the configurable system variable {PAGESIZE}. If an implementation has no
 1764 restrictions on size or alignment, it may specify a 1-byte page size.

1765 Memory locking guarantees the residence of portions of the address space. It is
 1766 implementation-defined whether locking memory guarantees fixed translation between virtual
 1767 addresses (as seen by the process) and physical addresses. Per-process memory locks are not
 1768 inherited across a *fork()*, and all memory locks owned by a process are unlocked upon *exec* or
 1769 process termination. Unmapping of an address range removes any memory locks established on
 1770 that address range by this process.

1771 2.8.3.2 Memory Mapped Files

1772 MF The functionality described in this section is dependent on support of the Memory Mapped Files
 1773 option (and the rest of this section is not further shaded for this option).

1774 Range memory mapping operations are defined in terms of pages. Implementations may
 1775 restrict the size and alignment of range mappings to be on page-size boundaries. The page size,
 1776 in bytes, is the value of the configurable system variable {PAGESIZE}. If an implementation has
 1777 no restrictions on size or alignment, it may specify a 1-byte page size.

1778 Memory mapped files provide a mechanism that allows a process to access files by directly
 1779 incorporating file data into its address space. Once a file is mapped into a process address space,
 1780 the data can be manipulated as memory. If more than one process maps a file, its contents are
 1781 shared among them. If the mappings allow shared write access, then data written into the
 1782 memory object through the address space of one process appears in the address spaces of all
 1783 processes that similarly map the same portion of the memory object.

1784 SHM Shared memory objects are named regions of storage that may be independent of the file system
 1785 and can be mapped into the address space of one or more processes to allow them to share the
 1786 associated memory.

1787 SHM An *unlink()* of a file or *shm_unlink()* of a shared memory object, while causing the removal of the
 1788 name, does not unmap any mappings established for the object. Once the name has been
 1789 removed, the contents of the memory object are preserved as long as it is referenced. The
 1790 memory object remains referenced as long as a process has the memory object open or has some
 1791 area of the memory object mapped.

1792 2.8.3.3 *Memory Protection*

1793 MPR MF The functionality described in this section is dependent on support of the Memory Protection
 1794 and Memory Mapped Files option (and the rest of this section is not further shaded for these
 1795 options).

1796 When an object is mapped, various application accesses to the mapped region may result in
 1797 signals. In this context, SIGBUS is used to indicate an error using the mapped object, and
 1798 SIGSEGV is used to indicate a protection violation or misuse of an address:

- 1799 • A mapping may be restricted to disallow some types of access.
- 1800 • Write attempts to memory that was mapped without write access, or any access to memory
 1801 mapped PROT_NONE, shall result in a SIGSEGV signal.
- 1802 • References to unmapped addresses shall result in a SIGSEGV signal.
- 1803 • Reference to whole pages within the mapping, but beyond the current length of the object,
 1804 shall result in a SIGBUS signal.
- 1805 • The size of the object is unaffected by access beyond the end of the object (even if a SIGBUS is
 1806 not generated).

1807 2.8.3.4 *Typed Memory Objects*

1808 TYM The functionality described in this section is dependent on support of the Typed Memory
 1809 Objects option (and the rest of this section is not further shaded for this option).

1810 Implementations may support the Typed Memory Objects option without supporting the
 1811 Memory Mapped Files option or the Shared Memory Objects option. Typed memory objects are
 1812 implementation-configurable named storage pools accessible from one or more processors in a
 1813 system, each via one or more ports, such as backplane buses, LANs, I/O channels, and so on.
 1814 Each valid combination of a storage pool and a port is identified through a name that is defined
 1815 at system configuration time, in an implementation-defined manner; the name may be
 1816 independent of the file system. Using this name, a typed memory object can be opened and
 1817 mapped into process address space. For a given storage pool and port, it is necessary to support
 1818 both dynamic allocation from the pool as well as mapping at an application-supplied offset
 1819 within the pool; when dynamic allocation has been performed, subsequent deallocation must be
 1820 supported. Lastly, accessing typed memory objects from different ports requires a method for
 1821 obtaining the offset and length of contiguous storage of a region of typed memory (dynamically
 1822 allocated or not); this allows typed memory to be shared among processes and/or processors
 1823 while being accessed from the desired port.

1824 2.8.4 *Process Scheduling*

1825 PS The functionality described in this section is dependent on support of the Process Scheduling
 1826 option (and the rest of this section is not further shaded for this option).

1827 *Scheduling Policies*

1828 The scheduling semantics described in this volume of IEEE Std 1003.1-2001 are defined in terms
 1829 of a conceptual model that contains a set of thread lists. No implementation structures are
 1830 necessarily implied by the use of this conceptual model. It is assumed that no time elapses
 1831 during operations described using this model, and therefore no simultaneous operations are
 1832 possible. This model discusses only processor scheduling for runnable threads, but it should be
 1833 noted that greatly enhanced predictability of realtime applications results if the sequencing of
 1834 other resources takes processor scheduling policy into account.

1835 There is, conceptually, one thread list for each priority. A runnable thread will be on the thread
 1836 list for that thread's priority. Multiple scheduling policies shall be provided. Each non-empty
 1837 thread list is ordered, contains a head as one end of its order, and a tail as the other. The purpose
 1838 of a scheduling policy is to define the allowable operations on this set of lists (for example,
 1839 moving threads between and within lists).

1840 Each process shall be controlled by an associated scheduling policy and priority. These
 1841 parameters may be specified by explicit application execution of the *sched_setscheduler()* or
 1842 *sched_setparam()* functions.

1843 Each thread shall be controlled by an associated scheduling policy and priority. These
 1844 parameters may be specified by explicit application execution of the *pthread_setschedparam()*
 1845 function.

1846 Associated with each policy is a priority range. Each policy definition shall specify the minimum
 1847 priority range for that policy. The priority ranges for each policy may but need not overlap the
 1848 priority ranges of other policies.

1849 A conforming implementation shall select the thread that is defined as being at the head of the
 1850 highest priority non-empty thread list to become a running thread, regardless of its associated
 1851 policy. This thread is then removed from its thread list.

1852 Four scheduling policies are specifically required. Other implementation-defined scheduling
 1853 policies may be defined. The following symbols are defined in the Base Definitions volume of
 1854 IEEE Std 1003.1-2001, <*sched.h*>:

1855 SCHED_FIFO First in, first out (FIFO) scheduling policy.

1856 SCHED_RR Round robin scheduling policy.

1857 SS SCHED_SPORADIC Sporadic server scheduling policy.

1858 SCHED_OTHER Another scheduling policy.

1859 The values of these symbols shall be distinct.

1860 SCHED_FIFO

1861 Conforming implementations shall include a scheduling policy called the FIFO scheduling
 1862 policy.

1863 Threads scheduled under this policy are chosen from a thread list that is ordered by the time its
 1864 threads have been on the list without being executed; generally, the head of the list is the thread
 1865 that has been on the list the longest time, and the tail is the thread that has been on the list the
 1866 shortest time.

1867 Under the SCHED_FIFO policy, the modification of the definitional thread lists is as follows:

- 1868 1. When a running thread becomes a preempted thread, it becomes the head of the thread list
 1869 for its priority.
- 1870 2. When a blocked thread becomes a runnable thread, it becomes the tail of the thread list for
 1871 its priority.
- 1872 3. When a running thread calls the *sched_setscheduler()* function, the process specified in the
 1873 function call is modified to the specified policy and the priority specified by the *param*
 1874 argument.
- 1875 4. When a running thread calls the *sched_setparam()* function, the priority of the process
 1876 specified in the function call is modified to the priority specified by the *param* argument.

5. When a running thread calls the *pthread_setschedparam()* function, the thread specified in the function call is modified to the specified policy and the priority specified by the *param* argument.
6. When a running thread calls the *pthread_setschedprio()* function, the thread specified in the function call is modified to the priority specified by the *prio* argument.
7. If a thread whose policy or priority has been modified other than by *pthread_setschedprio()* is a running thread or is runnable, it then becomes the tail of the thread list for its new priority.
8. If a thread whose policy or priority has been modified by *pthread_setschedprio()* is a running thread or is runnable, the effect on its position in the thread list depends on the direction of the modification, as follows:
 - a. If the priority is raised, the thread becomes the tail of the thread list.
 - b. If the priority is unchanged, the thread does not change position in the thread list.
 - c. If the priority is lowered, the thread becomes the head of the thread list.
9. When a running thread issues the *sched_yield()* function, the thread becomes the tail of the thread list for its priority.
10. At no other time is the position of a thread with this scheduling policy within the thread lists affected.

For this policy, valid priorities shall be within the range returned by the *sched_get_priority_max()* and *sched_get_priority_min()* functions when *SCHED_FIFO* is provided as the parameter. Conforming implementations shall provide a priority range of at least 32 priorities for this policy.

SCHED_RR

Conforming implementations shall include a scheduling policy called the “round robin” scheduling policy. This policy shall be identical to the *SCHED_FIFO* policy with the additional condition that when the implementation detects that a running thread has been executing as a running thread for a time period of the length returned by the *sched_rr_get_interval()* function or longer, the thread shall become the tail of its thread list and the head of that thread list shall be removed and made a running thread.

The effect of this policy is to ensure that if there are multiple *SCHED_RR* threads at the same priority, one of them does not monopolize the processor. An application should not rely only on the use of *SCHED_RR* to ensure application progress among multiple threads if the application includes threads using the *SCHED_FIFO* policy at the same or higher priority levels or *SCHED_RR* threads at a higher priority level.

A thread under this policy that is preempted and subsequently resumes execution as a running thread completes the unexpired portion of its round robin interval time period.

For this policy, valid priorities shall be within the range returned by the *sched_get_priority_max()* and *sched_get_priority_min()* functions when *SCHED_RR* is provided as the parameter. Conforming implementations shall provide a priority range of at least 32 priorities for this policy.

SCHED_SPORADIC

The functionality described in this section is dependent on support of the Process Sporadic Server or Thread Sporadic Server options (and the rest of this section is not further shaded for these options).

If `_POSIX_SPORADIC_SERVER` or `_POSIX_THREAD_SPORADIC_SERVER` is defined, the implementation shall include a scheduling policy identified by the value `SCHED_SPORADIC`.

The sporadic server policy is based primarily on two parameters: the replenishment period and the available execution capacity. The replenishment period is given by the `sched_ss_repl_period` member of the **sched_param** structure. The available execution capacity is initialized to the value given by the `sched_ss_init_budget` member of the same parameter. The sporadic server policy is identical to the `SCHED_FIFO` policy with some additional conditions that cause the thread's assigned priority to be switched between the values specified by the `sched_priority` and `sched_ss_low_priority` members of the **sched_param** structure.

The priority assigned to a thread using the sporadic server scheduling policy is determined in the following manner: if the available execution capacity is greater than zero and the number of pending replenishment operations is strictly less than `sched_ss_max_repl`, the thread is assigned the priority specified by `sched_priority`; otherwise, the assigned priority shall be `sched_ss_low_priority`. If the value of `sched_priority` is less than or equal to the value of `sched_ss_low_priority`, the results are undefined. When active, the thread shall belong to the thread list corresponding to its assigned priority level, according to the mentioned priority assignment. The modification of the available execution capacity and, consequently of the assigned priority, is done as follows:

1. When the thread at the head of the `sched_priority` list becomes a running thread, its execution time shall be limited to at most its available execution capacity, plus the resolution of the execution time clock used for this scheduling policy. This resolution shall be implementation-defined.
2. Each time the thread is inserted at the tail of the list associated with `sched_priority`—because as a blocked thread it became runnable with priority `sched_priority` or because a replenishment operation was performed—the time at which this operation is done is posted as the `activation_time`.
3. When the running thread with assigned priority equal to `sched_priority` becomes a preempted thread, it becomes the head of the thread list for its priority, and the execution time consumed is subtracted from the available execution capacity. If the available execution capacity would become negative by this operation, it shall be set to zero.
4. When the running thread with assigned priority equal to `sched_priority` becomes a blocked thread, the execution time consumed is subtracted from the available execution capacity, and a replenishment operation is scheduled, as described in 6 and 7. If the available execution capacity would become negative by this operation, it shall be set to zero.
5. When the running thread with assigned priority equal to `sched_priority` reaches the limit imposed on its execution time, it becomes the tail of the thread list for `sched_ss_low_priority`, the execution time consumed is subtracted from the available execution capacity (which becomes zero), and a replenishment operation is scheduled, as described in 6 and 7.
6. Each time a replenishment operation is scheduled, the amount of execution capacity to be replenished, `replenish_amount`, is set equal to the execution time consumed by the thread since the `activation_time`. The replenishment is scheduled to occur at `activation_time` plus `sched_ss_repl_period`. If the scheduled time obtained is before the current time, the

1964 replenishment operation is carried out immediately. Several replenishment operations may
 1965 be pending at the same time, each of which will be serviced at its respective scheduled
 1966 time. With the above rules, the number of replenishment operations simultaneously
 1967 pending for a given thread that is scheduled under the sporadic server policy shall not be
 1968 greater than *sched_ss_max_repl*.

1969 7. A replenishment operation consists of adding the corresponding *replenish_amount* to the
 1970 available execution capacity at the scheduled time. If, as a consequence of this operation,
 1971 the execution capacity would become larger than *sched_ss_initial_budget*, it shall be
 1972 rounded down to a value equal to *sched_ss_initial_budget*. Additionally, if the thread was
 1973 runnable or running, and had assigned priority equal to *sched_ss_low_priority*, then it
 1974 becomes the tail of the thread list for *sched_priority*.

1975 Execution time is defined in Section 2.2.2 (on page 14).

1976 For this policy, changing the value of a CPU-time clock via *clock_settime()* shall have no effect on
 1977 its behavior.

1978 For this policy, valid priorities shall be within the range returned by the *sched_get_priority_min()*
 1979 and *sched_get_priority_max()* functions when SCHED_SPORADIC is provided as the parameter.
 1980 Conforming implementations shall provide a priority range of at least 32 distinct priorities for
 1981 this policy.

1982 SCHED_OTHER

1983 Conforming implementations shall include one scheduling policy identified as SCHED_OTHER
 1984 (which may execute identically with either the FIFO or round robin scheduling policy). The
 1985 effect of scheduling threads with the SCHED_OTHER policy in a system in which other threads
 1986 SS are executing under SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC is implementation-
 1987 defined.

1988 This policy is defined to allow strictly conforming applications to be able to indicate in a
 1989 portable manner that they no longer need a realtime scheduling policy.

1990 For threads executing under this policy, the implementation shall use only priorities within the
 1991 range returned by the *sched_get_priority_max()* and *sched_get_priority_min()* functions when
 1992 SCHED_OTHER is provided as the parameter.

1993 2.8.5 Clocks and Timers

1994 TMR The functionality described in this section is dependent on support of the Timers option (and the
 1995 rest of this section is not further shaded for this option).

1996 The **<time.h>** header defines the types and manifest constants used by the timing facility.

1997 Time Value Specification Structures

1998 Many of the timing facility functions accept or return time value specifications. A time value
 1999 structure **timespec** specifies a single time value and includes at least the following members:

2000

2001

2002

2003

Member Type	Member Name	Description
time_t	<i>tv_sec</i>	Seconds.
long	<i>tv_nsec</i>	Nanoseconds.

2004 The *tv_nsec* member is only valid if greater than or equal to zero, and less than the number of
 2005 nanoseconds in a second (1 000 million). The time interval described by this structure is (*tv_sec* *
 2006 10⁹ + *tv_nsec*) nanoseconds.

A time value structure **itimerspec** specifies an initial timer value and a repetition interval for use by the per-process timer functions. This structure includes at least the following members:

Member Type	Member Name	Description
struct timespec	<i>it_interval</i>	Timer period.
struct timespec	<i>it_value</i>	Timer expiration.

If the value described by *it_value* is non-zero, it indicates the time to or time of the next timer expiration (for relative and absolute timer values, respectively). If the value described by *it_value* is zero, the timer shall be disarmed.

If the value described by *it_interval* is non-zero, it specifies an interval which shall be used in reloading the timer when it expires; that is, a periodic timer is specified. If the value described by *it_interval* is zero, the timer is disarmed after its next expiration; that is, a one-shot timer is specified.

Timer Event Notification Control Block

Per-process timers may be created that notify the process of timer expirations by queuing a realtime extended signal. The **sigevent** structure, defined in the Base Definitions volume of IEEE Std 1003.1-2001, <**signal.h**>, is used in creating such a timer. The **sigevent** structure contains the signal number and an application-specific data value which shall be used when notifying the calling process of timer expiration events.

Manifest Constants

The following constants are defined in the Base Definitions volume of IEEE Std 1003.1-2001, <**time.h**>:

CLOCK_REALTIME The identifier for the system-wide realtime clock.

TIMER_ABSTIME Flag indicating time is absolute with respect to the clock associated with a timer.

CLOCK_MONOTONIC The identifier for the system-wide monotonic clock, which is defined as a clock whose value cannot be set via *clock_settime()* and which cannot have backward clock jumps. The maximum possible clock jump is implementation-defined.

The maximum allowable resolution for **CLOCK_REALTIME** and **CLOCK_MONOTONIC** clocks and all time services based on these clocks is represented by **{_POSIX_CLOCKRES_MIN}** and shall be defined as 20 ms (1/50 of a second). Implementations may support smaller values of resolution for these clocks to provide finer granularity time bases. The actual resolution supported by an implementation for a specific clock is obtained using the *clock_getres()* function. If the actual resolution supported for a time service based on one of these clocks differs from the resolution supported for that clock, the implementation shall document this difference.

The minimum allowable maximum value for **CLOCK_REALTIME** and **CLOCK_MONOTONIC** clocks and all absolute time services based on them is the same as that defined by the ISO C standard for the **time_t** type. If the maximum value supported by a time service based on one of these clocks differs from the maximum value supported by that clock, the implementation shall document this difference.

2048 **Execution Time Monitoring**

2049 CPT If `_POSIX_CPUTIME` is defined, process CPU-time clocks shall be supported in addition to the
 2050 clocks described in **Manifest Constants** (on page 49).

2051 TCT If `_POSIX_THREAD_CPUTIME` is defined, thread CPU-time clocks shall be supported.

2052 CPT|TCT CPU-time clocks measure execution or CPU time, which is defined in the Base Definitions
 2053 volume of IEEE Std 1003.1-2001, Section 3.117, CPU Time (Execution Time). The mechanism
 2054 used to measure execution time is described in the Base Definitions volume of
 2055 IEEE Std 1003.1-2001, Section 4.9, Measurement of Execution Time.

2056 CPT If `_POSIX_CPUTIME` is defined, the following constant of the type `clockid_t` is defined in
 2057 `<time.h>`:

2058 `CLOCK_PROCESS_CPUTIME_ID`

2059 When this value of the type `clockid_t` is used in a `clock()` or `timer*()` function call, it is
 2060 interpreted as the identifier of the CPU-time clock associated with the process making the
 2061 function call.
 2062

2063 TCT If `_POSIX_THREAD_CPUTIME` is defined, the following constant of the type `clockid_t` is
 2064 defined in `<time.h>`:

2065 `CLOCK_THREAD_CPUTIME_ID`

2066 When this value of the type `clockid_t` is used in a `clock()` or `timer*()` function call, it is
 2067 interpreted as the identifier of the CPU-time clock associated with the thread making the
 2068 function call.

2069 **2.9 Threads**

2070 THR The functionality described in this section is dependent on support of the Threads option (and
 2071 the rest of this section is not further shaded for this option).

2072 This section defines functionality to support multiple flows of control, called “threads”, within a
 2073 process. For the definition of threads, see the Base Definitions volume of IEEE Std 1003.1-2001,
 2074 Section 3.393, Thread.

2075 The specific functional areas covered by threads and their scope include:

- 2076 • Thread management: the creation, control, and termination of multiple flows of control in the
 2077 same process under the assumption of a common shared address space
- 2078 • Synchronization primitives optimized for tightly coupled operation of multiple control flows
 2079 in a common, shared address space

2080 **2.9.1 Thread-Safety**

2081 All functions defined by this volume of IEEE Std 1003.1-2001 shall be thread-safe, except that the
 2082 following functions¹ need not be thread-safe.

2083 _____

2084 1. The functions in the table are not shaded to denote applicable options. Individual reference pages should be consulted.

2085	<i>asctime()</i>	<i>ecvt()</i>	<i>gethostent()</i>	<i>getutxline()</i>	<i>putc_unlocked()</i>
2086	<i>basename()</i>	<i>encrypt()</i>	<i>getlogin()</i>	<i>gmtime()</i>	<i>putchar_unlocked()</i>
2087	<i>catgets()</i>	<i>endgrent()</i>	<i>getnetbyaddr()</i>	<i>hcreate()</i>	<i>putenv()</i>
2088	<i>crypt()</i>	<i>endpwent()</i>	<i>getnetbyname()</i>	<i>hdestroy()</i>	<i>pututxline()</i>
2089	<i>ctime()</i>	<i>endtxent()</i>	<i>getnetent()</i>	<i>hsearch()</i>	<i>rand()</i>
2090	<i>dbm_clearerr()</i>	<i>fcvt()</i>	<i>getopt()</i>	<i>inet_ntoa()</i>	<i>readdir()</i>
2091	<i>dbm_close()</i>	<i>ftw()</i>	<i>getprotobyname()</i>	<i>l64a()</i>	<i>setenv()</i>
2092	<i>dbm_delete()</i>	<i>gcvt()</i>	<i>getprotobynumber()</i>	<i>lgamma()</i>	<i>setgrent()</i>
2093	<i>dbm_error()</i>	<i>getc_unlocked()</i>	<i>getprotoent()</i>	<i>lgammaf()</i>	<i>setkey()</i>
2094	<i>dbm_fetch()</i>	<i>getchar_unlocked()</i>	<i>getpwent()</i>	<i>lgammal()</i>	<i>setpwent()</i>
2095	<i>dbm_firstkey()</i>	<i>getdate()</i>	<i>getpwnam()</i>	<i>localeconv()</i>	<i>setutxent()</i>
2096	<i>dbm_nextkey()</i>	<i>getenv()</i>	<i>getpwuid()</i>	<i>localtime()</i>	<i>strerror()</i>
2097	<i>dbm_open()</i>	<i>getgrent()</i>	<i>getservbyname()</i>	<i>lrnd48()</i>	<i>strtok()</i>
2098	<i>dbm_store()</i>	<i>getgrgid()</i>	<i>getservbyport()</i>	<i>mrnd48()</i>	<i>ttynam()</i>
2099	<i>dirname()</i>	<i>getgrnam()</i>	<i>getservent()</i>	<i>nftw()</i>	<i>unsetenv()</i>
2100	<i>dllerror()</i>	<i>gethostbyaddr()</i>	<i>getutxent()</i>	<i>nl_langinfo()</i>	<i>wcstombs()</i>
2101	<i>drand48()</i>	<i>gethostbyname()</i>	<i>getutxid()</i>	<i>ptsname()</i>	<i>wctomb()</i>

2102 The *ctermid()* and *tmpnam()* functions need not be thread-safe if passed a NULL argument. The
 2103 *wctomb()* and *wcsrtombs()* functions need not be thread-safe if passed a NULL *ps* argument.

2104 Implementations shall provide internal synchronization as necessary in order to satisfy this
 2105 requirement.

2106 2.9.2 Thread IDs

2107 Although implementations may have thread IDs that are unique in a system, applications
 2108 should only assume that thread IDs are usable and unique within a single process. The effect of
 2109 calling any of the functions defined in this volume of IEEE Std 1003.1-2001 and passing as an
 2110 argument the thread ID of a thread from another process is unspecified. A conforming
 2111 implementation is free to reuse a thread ID after the thread terminates if it was created with the
 2112 *detachstate* attribute set to *PTHREAD_CREATE_DETACHED* or if *pthread_detach()* or
 2113 *pthread_join()* has been called for that thread. If a thread is detached, its thread ID is invalid for
 2114 use as an argument in a call to *pthread_detach()* or *pthread_join()*.

2115 2.9.3 Thread Mutexes

2116 A thread that has blocked shall not prevent any unblocked thread that is eligible to use the same
 2117 processing resources from eventually making forward progress in its execution. Eligibility for
 2118 processing resources is determined by the scheduling policy.

2119 A thread shall become the owner of a mutex, *m*, when one of the following occurs:

- 2120 • It returns successfully from *pthread_mutex_lock()* with *m* as the *mutex* argument.
- 2121 • It returns successfully from *pthread_mutex_trylock()* with *m* as the *mutex* argument.
- 2122 TMO • It returns successfully from *pthread_mutex_timedlock()* with *m* as the *mutex* argument.
- 2123 • It returns (successfully or not) from *pthread_cond_wait()* with *m* as the *mutex* argument
 2124 (except as explicitly indicated otherwise for certain errors).
- 2125 • It returns (successfully or not) from *pthread_cond_timedwait()* with *m* as the *mutex* argument
 2126 (except as explicitly indicated otherwise for certain errors).

2127 The thread shall remain the owner of *m* until one of the following occurs:

- 2128 • It executes `pthread_mutex_unlock()` with *m* as the *mutex* argument
 - 2129 • It blocks in a call to `pthread_cond_wait()` with *m* as the *mutex* argument.
 - 2130 • It blocks in a call to `pthread_cond_timedwait()` with *m* as the *mutex* argument.
- 2131 The implementation shall behave as if at all times there is at most one owner of any mutex.
- 2132 A thread that becomes the owner of a mutex is said to have “acquired” the mutex and the mutex
- 2133 is said to have become “locked”; when a thread gives up ownership of a mutex it is said to have
- 2134 “released” the mutex and the mutex is said to have become “unlocked”.

2135 **2.9.4 Thread Scheduling**

2136 TPS The functionality described in this section is dependent on support of the Thread Execution
 2137 Scheduling option (and the rest of this section is not further shaded for this option).

2138 **Thread Scheduling Attributes**

2139 In support of the scheduling function, threads have attributes which are accessed through the
 2140 `pthread_attr_t` thread creation attributes object.

2141 The *contentionscope* attribute defines the scheduling contention scope of the thread to be either
 2142 `PTHREAD_SCOPE_PROCESS` or `PTHREAD_SCOPE_SYSTEM`.

2143 The *inheritsched* attribute specifies whether a newly created thread is to inherit the scheduling
 2144 attributes of the creating thread or to have its scheduling values set according to the other
 2145 scheduling attributes in the `pthread_attr_t` object.

2146 The *schedpolicy* attribute defines the scheduling policy for the thread. The *schedparam* attribute
 2147 defines the scheduling parameters for the thread. The interaction of threads having different
 2148 policies within a process is described as part of the definition of those policies.

2149 If the Thread Execution Scheduling option is defined, and the *schedpolicy* attribute specifies one
 2150 of the priority-based policies defined under this option, the *schedparam* attribute contains the
 2151 scheduling priority of the thread. A conforming implementation ensures that the priority value
 2152 in *schedparam* is in the range associated with the scheduling policy when the thread attributes
 2153 object is used to create a thread, or when the scheduling attributes of a thread are dynamically
 2154 modified. The meaning of the priority value in *schedparam* is the same as that of *priority*.

2155 TSP If `_POSIX_THREAD_SPORADIC_SERVER` is defined, the *schedparam* attribute supports four
 2156 new members that are used for the sporadic server scheduling policy. These members are
 2157 *sched_ss_low_priority*, *sched_ss_repl_period*, *sched_ss_init_budget*, and *sched_ss_max_repl*. The
 2158 meaning of these attributes is the same as in the definitions that appear under Section 2.8.4 (on
 2159 page 44).

2160 When a process is created, its single thread has a scheduling policy and associated attributes
 2161 equal to the process’ policy and attributes. The default scheduling contention scope value is
 2162 implementation-defined. The default values of other scheduling attributes are implementation-
 2163 defined.

Thread Scheduling Contention Scope

The scheduling contention scope of a thread defines the set of threads with which the thread competes for use of the processing resources. The scheduling operation selects at most one thread to execute on each processor at any point in time and the thread's scheduling attributes (for example, *priority*), whether under process scheduling contention scope or system scheduling contention scope, are the parameters used to determine the scheduling decision.

The scheduling contention scope, in the context of scheduling a mixed scope environment, affects threads as follows:

- A thread created with `PTHREAD_SCOPE_SYSTEM` scheduling contention scope contends for resources with all other threads in the same scheduling allocation domain relative to their system scheduling attributes. The system scheduling attributes of a thread created with `PTHREAD_SCOPE_SYSTEM` scheduling contention scope are the scheduling attributes with which the thread was created. The system scheduling attributes of a thread created with `PTHREAD_SCOPE_PROCESS` scheduling contention scope are the implementation-defined mapping into system attribute space of the scheduling attributes with which the thread was created.
- Threads created with `PTHREAD_SCOPE_PROCESS` scheduling contention scope contend directly with other threads within their process that were created with `PTHREAD_SCOPE_PROCESS` scheduling contention scope. The contention is resolved based on the threads' scheduling attributes and policies. It is unspecified how such threads are scheduled relative to threads in other processes or threads with `PTHREAD_SCOPE_SYSTEM` scheduling contention scope.
- Conforming implementations shall support the `PTHREAD_SCOPE_PROCESS` scheduling contention scope, the `PTHREAD_SCOPE_SYSTEM` scheduling contention scope, or both.

Scheduling Allocation Domain

Implementations shall support scheduling allocation domains containing one or more processors. It should be noted that the presence of multiple processors does not automatically indicate a scheduling allocation domain size greater than one. Conforming implementations on multi-processors may map all or any subset of the CPUs to one or multiple scheduling allocation domains, and could define these scheduling allocation domains on a per-thread, per-process, or per-system basis, depending on the types of applications intended to be supported by the implementation. The scheduling allocation domain is independent of scheduling contention scope, as the scheduling contention scope merely defines the set of threads with which a thread contends for processor resources, while scheduling allocation domain defines the set of processors for which it contends. The semantics of how this contention is resolved among threads for processors is determined by the scheduling policies of the threads.

The choice of scheduling allocation domain size and the level of application control over scheduling allocation domains is implementation-defined. Conforming implementations may change the size of scheduling allocation domains and the binding of threads to scheduling allocation domains at any time.

For application threads with scheduling allocation domains of size equal to one, the scheduling rules defined for `SCHED_FIFO` and `SCHED_RR` shall be used; see **Scheduling Policies** (on page 44). All threads with system scheduling contention scope, regardless of the processes in which they reside, compete for the processor according to their priorities. Threads with process scheduling contention scope compete only with other threads with process scheduling contention scope within their process.

2210 For application threads with scheduling allocation domains of size greater than one, the rules
 2211 TSP defined for SCHED_FIFO, SCHED_RR, and SCHED_SPORADIC shall be used in an
 2212 implementation-defined manner. Each thread with system scheduling contention scope
 2213 competes for the processors in its scheduling allocation domain in an implementation-defined
 2214 manner according to its priority. Threads with process scheduling contention scope are
 2215 scheduled relative to other threads within the same scheduling contention scope in the process.

2216 TSP If _POSIX_THREAD_SPORADIC_SERVER is defined, the rules defined for SCHED_SPORADIC
 2217 in **Scheduling Policies** (on page 44) shall be used in an implementation-defined manner for
 2218 application threads whose scheduling allocation domain size is greater than one.

2219 **Scheduling Documentation**

2220 If _POSIX_PRIORITY_SCHEDULING is defined, then any scheduling policies beyond
 2221 TSP SCHED_OTHER, SCHED_FIFO, SCHED_RR, and SCHED_SPORADIC, as well as the effects of
 2222 the scheduling policies indicated by these other values, and the attributes required in order to
 2223 support such a policy, are implementation-defined. Furthermore, the implementation shall
 2224 document the effect of all processor scheduling allocation domain values supported for these
 2225 policies.

2226 **2.9.5 Thread Cancellation**

2227 The thread cancellation mechanism allows a thread to terminate the execution of any other
 2228 thread in the process in a controlled manner. The target thread (that is, the one that is being
 2229 canceled) is allowed to hold cancellation requests pending in a number of ways and to perform
 2230 application-specific cleanup processing when the notice of cancellation is acted upon.

2231 Cancellation is controlled by the cancellation control functions. Each thread maintains its own
 2232 cancelability state. Cancellation may only occur at cancellation points or when the thread is
 2233 asynchronously cancelable.

2234 The thread cancellation mechanism described in this section depends upon programs having set
 2235 *deferred* cancelability state, which is specified as the default. Applications shall also carefully
 2236 follow static lexical scoping rules in their execution behavior. For example, use of *setjmp()*,
 2237 *return*, *goto*, and so on, to leave user-defined cancellation scopes without doing the necessary
 2238 scope pop operation results in undefined behavior.

2239 Use of asynchronous cancelability while holding resources which potentially need to be released
 2240 may result in resource loss. Similarly, cancellation scopes may only be safely manipulated
 2241 (pushed and popped) when the thread is in the *deferred* or *disabled* cancelability states.

2242 **2.9.5.1 Cancelability States**

2243 The cancelability state of a thread determines the action taken upon receipt of a cancellation
 2244 request. The thread may control cancellation in a number of ways.

2245 Each thread maintains its own cancelability state, which may be encoded in two bits:

- 2246 1. Cancelability-Enable: When cancelability is PTHREAD_CANCEL_DISABLE (as defined in
 2247 the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>), cancellation requests
 2248 against the target thread are held pending. By default, cancelability is set to
 2249 PTHREAD_CANCEL_ENABLE (as defined in <pthread.h>).
- 2250 2. Cancelability Type: When cancelability is enabled and the cancelability type is
 2251 PTHREAD_CANCEL_ASYNCHRONOUS (as defined in <pthread.h>), new or pending
 2252 cancellation requests may be acted upon at any time. When cancelability is enabled and the
 2253 cancelability type is PTHREAD_CANCEL_DEFERRED (as defined in <pthread.h>),

2254 cancellation requests are held pending until a cancellation point (see below) is reached. If
 2255 cancelability is disabled, the setting of the cancelability type has no immediate effect as all
 2256 cancellation requests are held pending; however, once cancelability is enabled again the
 2257 new type is in effect. The cancelability type is PTHREAD_CANCEL_DEFERRED in all
 2258 newly created threads including the thread in which *main()* was first invoked.

2259 2.9.5.2 Cancellation Points

2260 Cancellation points shall occur when a thread is executing the following functions:

2261	<i>accept()</i>	<i>mq_timedsend()</i>	<i>putpmsg()</i>	<i>sigsuspend()</i>
2262	<i>aio_suspend()</i>	<i>msgrcv()</i>	<i>pwrite()</i>	<i>sigtimedwait()</i>
2263	<i>clock_nanosleep()</i>	<i>msgsnd()</i>	<i>read()</i>	<i>sigwait()</i>
2264	<i>close()</i>	<i>msync()</i>	<i>readv()</i>	<i>sigwaitinfo()</i>
2265	<i>connect()</i>	<i>nanosleep()</i>	<i>recv()</i>	<i>sleep()</i>
2266	<i>creat()</i>	<i>open()</i>	<i>recvfrom()</i>	<i>system()</i>
2267	<i>fcntl()</i> ²	<i>pause()</i>	<i>recvmsg()</i>	<i>tcdrain()</i>
2268	<i>fsync()</i>	<i>poll()</i>	<i>select()</i>	<i>usleep()</i>
2269	<i>getmsg()</i>	<i>pread()</i>	<i>sem_timedwait()</i>	<i>wait()</i>
2270	<i>getpmsg()</i>	<i>pthread_cond_timedwait()</i>	<i>sem_wait()</i>	<i>waitid()</i>
2271	<i>lockf()</i>	<i>pthread_cond_wait()</i>	<i>send()</i>	<i>waitpid()</i>
2272	<i>mq_receive()</i>	<i>pthread_join()</i>	<i>sendmsg()</i>	<i>write()</i>
2273	<i>mq_send()</i>	<i>pthread_testcancel()</i>	<i>sendto()</i>	<i>writew()</i>
2274	<i>mq_timedreceive()</i>	<i>putmsg()</i>	<i>sigpause()</i>	

2275 _____

2276 2. When the *cmd* argument is F_SETLK.

2277 A cancelation point may also occur when a thread is executing the following functions:

2278	<i>catclose()</i>	<i>ftell()</i>	<i>getwc()</i>	<i>pthread_rwlock_wrlock()</i>
2279	<i>catgets()</i>	<i>ftello()</i>	<i>getwchar()</i>	<i>putc()</i>
2280	<i>catopen()</i>	<i>ftw()</i>	<i>getwd()</i>	<i>putc_unlocked()</i>
2281	<i>closedir()</i>	<i>fwprintf()</i>	<i>glob()</i>	<i>putchar()</i>
2282	<i>closelog()</i>	<i>fwrite()</i>	<i>iconv_close()</i>	<i>putchar_unlocked()</i>
2283	<i>ctermid()</i>	<i>fwscanf()</i>	<i>iconv_open()</i>	<i>puts()</i>
2284	<i>dbm_close()</i>	<i>getc()</i>	<i>ioctl()</i>	<i>pututxline()</i>
2285	<i>dbm_delete()</i>	<i>getc_unlocked()</i>	<i>lseek()</i>	<i>putwc()</i>
2286	<i>dbm_fetch()</i>	<i>getchar()</i>	<i>mkstemp()</i>	<i>putwchar()</i>
2287	<i>dbm_nextkey()</i>	<i>getchar_unlocked()</i>	<i>nftw()</i>	<i>readdir()</i>
2288	<i>dbm_open()</i>	<i>getcwd()</i>	<i>opendir()</i>	<i>readdir_r()</i>
2289	<i>dbm_store()</i>	<i>getdate()</i>	<i>openlog()</i>	<i>remove()</i>
2290	<i>dlclose()</i>	<i>getgrent()</i>	<i>pclose()</i>	<i>rename()</i>
2291	<i>dlopen()</i>	<i>getgrgid()</i>	<i>perror()</i>	<i>rewind()</i>
2292	<i>endgrent()</i>	<i>getgrgid_r()</i>	<i>popen()</i>	<i>rewinddir()</i>
2293	<i>endhostent()</i>	<i>getgrnam()</i>	<i>posix_fadvise()</i>	<i>scanf()</i>
2294	<i>endnetent()</i>	<i>getgrnam_r()</i>	<i>posix_fallocate()</i>	<i>seekdir()</i>
2295	<i>endprotoent()</i>	<i>gethostbyaddr()</i>	<i>posix_madvise()</i>	<i>semop()</i>
2296	<i>endpwent()</i>	<i>gethostbyname()</i>	<i>posix_spawn()</i>	<i>setgrent()</i>
2297	<i>endservent()</i>	<i>gethostent()</i>	<i>posix_spawnnp()</i>	<i>sethostent()</i>
2298	<i>endutxent()</i>	<i>gethostname()</i>	<i>posix_trace_clear()</i>	<i>setnetent()</i>
2299	<i>fclose()</i>	<i>getlogin()</i>	<i>posix_trace_close()</i>	<i>setprotoent()</i>
2300	<i>fcntl()</i> ³	<i>getlogin_r()</i>	<i>posix_trace_create()</i>	<i>setpwent()</i>
2301	<i>fflush()</i>	<i>getnetbyaddr()</i>	<i>posix_trace_create_withlog()</i>	<i>setservent()</i>
2302	<i>fgetc()</i>	<i>getnetbyname()</i>	<i>posix_trace_eventtypelist_getnext_id()</i>	<i>setutxent()</i>
2303	<i>fgetpos()</i>	<i>getnetent()</i>	<i>posix_trace_eventtypelist_rewind()</i>	<i>strerror()</i>
2304	<i>fgets()</i>	<i>getprotobynname()</i>	<i>posix_trace_flush()</i>	<i>syslog()</i>
2305	<i>fgetwc()</i>	<i>getprotobynumber()</i>	<i>posix_trace_get_attr()</i>	<i>tmpfile()</i>
2306	<i>fgetws()</i>	<i>getprotoent()</i>	<i>posix_trace_get_filter()</i>	<i>tmpnam()</i>
2307	<i>fopen()</i>	<i>getpwent()</i>	<i>posix_trace_get_status()</i>	<i>ttyname()</i>
2308	<i>fprintf()</i>	<i>getpwnam()</i>	<i>posix_trace_getnext_event()</i>	<i>ttyname_r()</i>
2309	<i>fputc()</i>	<i>getpwnam_r()</i>	<i>posix_trace_open()</i>	<i>ungetc()</i>
2310	<i>fputs()</i>	<i>getpwuid()</i>	<i>posix_trace_rewind()</i>	<i>ungetwc()</i>
2311	<i>fputwc()</i>	<i>getpwuid_r()</i>	<i>posix_trace_set_filter()</i>	<i>unlink()</i>
2312	<i>fputws()</i>	<i>gets()</i>	<i>posix_trace_shutdown()</i>	<i>vfprintf()</i>
2313	<i>fread()</i>	<i>getservbyname()</i>	<i>posix_trace_timedgetnext_event()</i>	<i>vfwprintf()</i>
2314	<i>freopen()</i>	<i>getservbyport()</i>	<i>posix_typed_mem_open()</i>	<i>vprintf()</i>
2315	<i>fscanf()</i>	<i>getservent()</i>	<i>printf()</i>	<i>vwprintf()</i>
2316	<i>fseek()</i>	<i>getutxent()</i>	<i>pthread_rwlock_rdlock()</i>	<i>wprintf()</i>
2317	<i>fseeko()</i>	<i>getutxid()</i>	<i>pthread_rwlock_timedrdlock()</i>	<i>wscanf()</i>
2318	<i>fsetpos()</i>	<i>getutxline()</i>	<i>pthread_rwlock_timedwrlock()</i>	

2319 An implementation shall not introduce cancelation points into any other functions specified in
 2320 this volume of IEEE Std 1003.1-2001.

2321 _____
 2322 3. For any value of the *cmd* argument.

2323 The side effects of acting upon a cancelation request while suspended during a call of a function
 2324 are the same as the side effects that may be seen in a single-threaded program when a call to a
 2325 function is interrupted by a signal and the given function returns [EINTR]. Any such side effects
 2326 occur before any cancelation cleanup handlers are called.

2327 Whenever a thread has cancelability enabled and a cancelation request has been made with that
 2328 thread as the target, and the thread then calls any function that is a cancelation point (such as
 2329 *pthread_testcancel()* or *read()*), the cancelation request shall be acted upon before the function
 2330 returns. If a thread has cancelability enabled and a cancelation request is made with the thread
 2331 as a target while the thread is suspended at a cancelation point, the thread shall be awakened
 2332 and the cancelation request shall be acted upon. However, if the thread is suspended at a
 2333 cancelation point and the event for which it is waiting occurs before the cancelation request is
 2334 acted upon, it is unspecified whether the cancelation request is acted upon or whether the
 2335 cancelation request remains pending and the thread resumes normal execution.

2336 2.9.5.3 Thread Cancelation Cleanup Handlers

2337 Each thread maintains a list of cancelation cleanup handlers. The programmer uses the
 2338 *pthread_cleanup_push()* and *pthread_cleanup_pop()* functions to place routines on and remove
 2339 routines from this list.

2340 When a cancelation request is acted upon, the routines in the list are invoked one by one in LIFO
 2341 sequence; that is, the last routine pushed onto the list (Last In) is the first to be invoked (First
 2342 Out). The thread invokes the cancelation cleanup handler with cancelation disabled until the last
 2343 cancelation cleanup handler returns. When the cancelation cleanup handler for a scope is
 2344 invoked, the storage for that scope remains valid. If the last cancelation cleanup handler returns,
 2345 thread execution is terminated and a status of PTHREAD_CANCELED is made available to any
 2346 threads joining with the target. The symbolic constant PTHREAD_CANCELED expands to a
 2347 constant expression of type (**void ***) whose value matches no pointer to an object in memory nor
 2348 the value NULL.

2349 The cancelation cleanup handlers are also invoked when the thread calls *pthread_exit()*.

2350 A side effect of acting upon a cancelation request while in a condition variable wait is that the
 2351 mutex is re-acquired before calling the first cancelation cleanup handler. In addition, the thread
 2352 is no longer considered to be waiting for the condition and the thread shall not have consumed
 2353 any pending condition signals on the condition.

2354 A cancelation cleanup handler cannot exit via *longjmp()* or *siglongjmp()*.

2355 2.9.5.4 Async-Cancel Safety

2356 The *pthread_cancel()*, *pthread_setcancelstate()*, and *pthread_setcanceltype()* functions are defined to
 2357 be async-cancel safe.

2358 No other functions in this volume of IEEE Std 1003.1-2001 are required to be async-cancel-safe.

2359 2.9.6 Thread Read-Write Locks

2360 Multiple readers, single writer (read-write) locks allow many threads to have simultaneous
 2361 read-only access to data while allowing only one thread to have exclusive write access at any
 2362 given time. They are typically used to protect data that is read more frequently than it is
 2363 changed.

2364 One or more readers acquire read access to the resource by performing a read lock operation on
 2365 the associated read-write lock. A writer acquires exclusive write access by performing a write
 2366 lock operation. Basically, all readers exclude any writers and a writer excludes all readers and
 2367 any other writers.

2368 A thread that has blocked on a read-write lock (for example, has not yet returned from a
 2369 *pthread_rwlock_rdlock()* or *pthread_rwlock_wrlock()* call) shall not prevent any unblocked thread
 2370 that is eligible to use the same processing resources from eventually making forward progress in
 2371 its execution. Eligibility for processing resources shall be determined by the scheduling policy.

2372 Read-write locks can be used to synchronize threads in the current process and other processes if
 2373 they are allocated in memory that is writable and shared among the cooperating processes and
 2374 have been initialized for this behavior.

2375 2.9.7 Thread Interactions with Regular File Operations

2376 All of the functions *chmod()*, *close()*, *fchmod()*, *fcntl()*, *fstat()*, *ftruncate()*, *lseek()*, *open()*, *read()*,
 2377 *readlink()*, *stat()*, *symlink()*, and *write()* shall be atomic with respect to each other in the effects
 2378 specified in IEEE Std 1003.1-2001 when they operate on regular files. If two threads each call one
 2379 of these functions, each call shall either see all of the specified effects of the other call, or none of
 2380 them.

2381 2.10 Sockets

2382 A socket is an endpoint for communication using the facilities described in this section. A socket
 2383 is created with a specific socket type, described in Section 2.10.6 (on page 59), and is associated
 2384 with a specific protocol, detailed in Section 2.10.3 (on page 59). A socket is accessed via a file
 2385 descriptor obtained when the socket is created.

2386 2.10.1 Address Families

2387 All network protocols are associated with a specific address family. An address family provides
 2388 basic services to the protocol implementation to allow it to function within a specific network
 2389 environment. These services may include packet fragmentation and reassembly, routing,
 2390 addressing, and basic transport. An address family is normally comprised of a number of
 2391 protocols, one per socket type. Each protocol is characterized by an abstract socket type. It is not
 2392 required that an address family support all socket types. An address family may contain
 2393 multiple protocols supporting the same socket abstraction.

2394 Section 2.10.17 (on page 66), Section 2.10.19 (on page 67), and Section 2.10.20 (on page 67),
 2395 respectively, describe the use of sockets for local UNIX connections, for Internet protocols based
 2396 on IPv4, and for Internet protocols based on IPv6.

2397 2.10.2 Addressing

2398 An address family defines the format of a socket address. All network addresses are described
 2399 using a general structure, called a **sockaddr**, as defined in the Base Definitions volume of
 2400 IEEE Std 1003.1-2001, <sys/socket.h>. However, each address family imposes finer and more
 2401 specific structure, generally defining a structure with fields specific to the address family. The
 2402 field *sa_family* in the **sockaddr** structure contains the address family identifier, specifying the
 2403 format of the *sa_data* area. The size of the *sa_data* area is unspecified.

2404 2.10.3 Protocols

2405 A protocol supports one of the socket abstractions detailed in Section 2.10.6. Selecting a protocol
 2406 involves specifying the address family, socket type, and protocol number to the *socket()*
 2407 function. Certain semantics of the basic socket abstractions are protocol-specific. All protocols
 2408 are expected to support the basic model for their particular socket type, but may, in addition,
 2409 provide non-standard facilities or extensions to a mechanism.

2410 2.10.4 Routing

2411 Sockets provides packet routing facilities. A routing information database is maintained, which
 2412 is used in selecting the appropriate network interface when transmitting packets.

2413 2.10.5 Interfaces

2414 Each network interface in a system corresponds to a path through which messages can be sent
 2415 and received. A network interface usually has a hardware device associated with it, though
 2416 certain interfaces such as the loopback interface, do not.

2417 2.10.6 Socket Types

2418 A socket is created with a specific type, which defines the communication semantics and which
 2419 RS allows the selection of an appropriate communication protocol. Four types are defined:
 2420 **SOCK_RAW**, **SOCK_STREAM**, **SOCK_SEQPACKET**, and **SOCK_DGRAM**. Implementations
 2421 may specify additional socket types.

2422 The **SOCK_STREAM** socket type provides reliable, sequenced, full-duplex octet streams
 2423 between the socket and a peer to which the socket is connected. A socket of type
 2424 **SOCK_STREAM** must be in a connected state before any data may be sent or received. Record
 2425 boundaries are not maintained; data sent on a stream socket using output operations of one size
 2426 may be received using input operations of smaller or larger sizes without loss of data. Data may
 2427 be buffered; successful return from an output function does not imply that the data has been
 2428 delivered to the peer or even transmitted from the local system. If data cannot be successfully
 2429 transmitted within a given time then the connection is considered broken, and subsequent
 2430 operations shall fail. A SIGPIPE signal is raised if a thread sends on a broken stream (one that is
 2431 no longer connected). Support for an out-of-band data transmission facility is protocol-specific.

2432 The **SOCK_SEQPACKET** socket type is similar to the **SOCK_STREAM** type, and is also
 2433 connection-oriented. The only difference between these types is that record boundaries are
 2434 maintained using the **SOCK_SEQPACKET** type. A record can be sent using one or more output
 2435 operations and received using one or more input operations, but a single operation never
 2436 transfers parts of more than one record. Record boundaries are visible to the receiver via the
 2437 **MSG_EOR** flag in the received message flags returned by the *recvmsg()* function. It is protocol-
 2438 specific whether a maximum record size is imposed.

2439 The **SOCK_DGRAM** socket type supports connectionless data transfer which is not necessarily
 2440 acknowledged or reliable. Datagrams may be sent to the address specified (possibly multicast or

2441 broadcast) in each output operation, and incoming datagrams may be received from multiple
 2442 sources. The source address of each datagram is available when receiving the datagram. An
 2443 application may also pre-specify a peer address, in which case calls to output functions shall
 2444 send to the pre-specified peer. If a peer has been specified, only datagrams from that peer shall
 2445 be received. A datagram must be sent in a single output operation, and must be received in a
 2446 single input operation. The maximum size of a datagram is protocol-specific; with some
 2447 protocols, the limit is implementation-defined. Output datagrams may be buffered within the
 2448 system; thus, a successful return from an output function does not guarantee that a datagram is
 2449 actually sent or received. However, implementations should attempt to detect any errors
 2450 possible before the return of an output function, reporting any error by an unsuccessful return
 2451 value.

2452 RS The SOCK_RAW socket type is similar to the SOCK_DGRAM type. It differs in that it is
 2453 normally used with communication providers that underlie those used for the other socket
 2454 types. For this reason, the creation of a socket with type SOCK_RAW shall require appropriate
 2455 privilege. The format of datagrams sent and received with this socket type generally include
 2456 specific protocol headers, and the formats are protocol-specific and implementation-defined.

2457 2.10.7 Socket I/O Mode

2458 The I/O mode of a socket is described by the O_NONBLOCK file status flag which pertains to
 2459 the open file description for the socket. This flag is initially off when a socket is created, but may
 2460 be set and cleared by the use of the F_SETFL command of the *fcntl()* function.

2461 When the O_NONBLOCK flag is set, functions that would normally block until they are
 2462 complete shall either return immediately with an error, or shall complete asynchronously to the
 2463 execution of the calling process. Data transfer operations (the *read()*, *write()*, *send()*, and *recv()*
 2464 functions) shall complete immediately, transfer only as much as is available, and then return
 2465 without blocking, or return an error indicating that no transfer could be made without blocking.
 2466 The *connect()* function initiates a connection and shall return without blocking when
 2467 O_NONBLOCK is set; it shall return the error [EINPROGRESS] to indicate that the connection
 2468 was initiated successfully, but that it has not yet completed.

2469 2.10.8 Socket Owner

2470 The owner of a socket is unset when a socket is created. The owner may be set to a process ID or
 2471 process group ID using the F_SETOWN command of the *fcntl()* function.

2472 2.10.9 Socket Queue Limits

2473 The transmit and receive queue sizes for a socket are set when the socket is created. The default
 2474 sizes used are both protocol-specific and implementation-defined. The sizes may be changed
 2475 using the *setsockopt()* function.

2476 2.10.10 Pending Error

2477 Errors may occur asynchronously, and be reported to the socket in response to input from the
 2478 network protocol. The socket stores the pending error to be reported to a user of the socket at the
 2479 next opportunity. The error is returned in response to a subsequent *send()*, *recv()*, or *getsockopt()*
 2480 operation on the socket, and the pending error is then cleared.

2.10.11 Socket Receive Queue

A socket has a receive queue that buffers data when it is received by the system until it is removed by a receive call. Depending on the type of the socket and the communication provider, the receive queue may also contain ancillary data such as the addressing and other protocol data associated with the normal data in the queue, and may contain out-of-band or expedited data. The limit on the queue size includes any normal, out-of-band data, datagram source addresses, and ancillary data in the queue. The description in this section applies to all sockets, even though some elements cannot be present in some instances.

The contents of a receive buffer are logically structured as a series of data segments with associated ancillary data and other information. A data segment may contain normal data or out-of-band data, but never both. A data segment may complete a record if the protocol supports records (always true for types SOCK_SEQPACKET and SOCK_DGRAM). A record may be stored as more than one segment; the complete record might never be present in the receive buffer at one time, as a portion might already have been returned to the application, and another portion might not yet have been received from the communications provider. A data segment may contain ancillary protocol data, which is logically associated with the segment. Ancillary data is received as if it were queued along with the first normal data octet in the segment (if any). A segment may contain ancillary data only, with no normal or out-of-band data. For the purposes of this section, a datagram is considered to be a data segment that terminates a record, and that includes a source address as a special type of ancillary data. Data segments are placed into the queue as data is delivered to the socket by the protocol. Normal data segments are placed at the end of the queue as they are delivered. If a new segment contains the same type of data as the preceding segment and includes no ancillary data, and if the preceding segment does not terminate a record, the segments are logically merged into a single segment.

The receive queue is logically terminated if an end-of-file indication has been received or a connection has been terminated. A segment shall be considered to be terminated if another segment follows it in the queue, if the segment completes a record, or if an end-of-file or other connection termination has been reported. The last segment in the receive queue shall also be considered to be terminated while the socket has a pending error to be reported.

A receive operation shall never return data or ancillary data from more than one segment.

2.10.12 Socket Out-of-Band Data State

The handling of received out-of-band data is protocol-specific. Out-of-band data may be placed in the socket receive queue, either at the end of the queue or before all normal data in the queue. In this case, out-of-band data is returned to an application program by a normal receive call. Out-of-band data may also be queued separately rather than being placed in the socket receive queue, in which case it shall be returned only in response to a receive call that requests out-of-band data. It is protocol-specific whether an out-of-band data mark is placed in the receive queue to demarcate data preceding the out-of-band data and following the out-of-band data. An out-of-band data mark is logically an empty data segment that cannot be merged with other segments in the queue. An out-of-band data mark is never returned in response to an input operation. The *socketmark()* function can be used to test whether an out-of-band data mark is the first element in the queue. If an out-of-band data mark is the first element in the queue when an input function is called without the MSG_PEEK option, the mark is removed from the queue and the following data (if any) is processed as if the mark had not been present.

2526 2.10.13 Connection Indication Queue

2527 Sockets that are used to accept incoming connections maintain a queue of outstanding
2528 connection indications. This queue is a list of connections that are awaiting acceptance by the
2529 application; see *listen()*.

2530 2.10.14 Signals

2531 One category of event at the socket interface is the generation of signals. These signals report
2532 protocol events or process errors relating to the state of the socket. The generation or delivery of
2533 a signal does not change the state of the socket, although the generation of the signal may have
2534 been caused by a state change.

2535 The SIGPIPE signal shall be sent to a thread that attempts to send data on a socket that is no
2536 longer able to send. In addition, the send operation fails with the error [EPIPE].

2537 If a socket has an owner, the SIGURG signal is sent to the owner of the socket when it is notified
2538 of expedited or out-of-band data. The socket state at this time is protocol-dependent, and the
2539 status of the socket is specified in Section 2.10.17 (on page 66), Section 2.10.19 (on page 67), and
2540 Section 2.10.20 (on page 67). Depending on the protocol, the expedited data may or may not
2541 have arrived at the time of signal generation.

2542 2.10.15 Asynchronous Errors

2543 If any of the following conditions occur asynchronously for a socket, the corresponding value
2544 listed below shall become the pending error for the socket:

2545 [ECONNABORTED]

2546 The connection was aborted locally.

2547 [ECONNREFUSED]

2548 For a connection-mode socket attempting a non-blocking connection, the attempt to connect
2549 was forcefully rejected. For a connectionless-mode socket, an attempt to deliver a datagram
2550 was forcefully rejected.

2551 [ECONNRESET]

2552 The peer has aborted the connection.

2553 [EHOSTDOWN]

2554 The destination host has been determined to be down or disconnected.

2555 [EHOSTUNREACH]

2556 The destination host is not reachable.

2557 [EMSGSIZE]

2558 For a connectionless-mode socket, the size of a previously sent datagram prevented
2559 delivery.

2560 [ENETDOWN]

2561 The local network connection is not operational.

2562 [ENETRESET]

2563 The connection was aborted by the network.

2564 [ENETUNREACH]

2565 The destination network is not reachable.

2566 **2.10.16 Use of Options**

2567 There are a number of socket options which either specialize the behavior of a socket or provide
 2568 useful information. These options may be set at different protocol levels and are always present
 2569 at the uppermost “socket” level.

2570 Socket options are manipulated by two functions, *getsockopt()* and *setsockopt()*. These functions
 2571 allow an application program to customize the behavior and characteristics of a socket to
 2572 provide the desired effect.

2573 All of the options have default values. The type and meaning of these values is defined by the
 2574 protocol level to which they apply. Instead of using the default values, an application program
 2575 may choose to customize one or more of the options. However, in the bulk of cases, the default
 2576 values are sufficient for the application.

2577 Some of the options are used to enable or disable certain behavior within the protocol modules
 2578 (for example, turn on debugging) while others may be used to set protocol-specific information
 2579 (for example, IP time-to-live on all the application's outgoing packets). As each of the options is
 2580 introduced, its effect on the underlying protocol modules is described.

2581 Table 2-1 shows the value for the socket level.

2582 **Table 2-1** Value of Level for Socket Options

Name	Description
SOL_SOCKET	Options are intended for the sockets level.

2585 Table 2-2 (on page 64) lists those options present at the socket level; that is, when the *level*
 2586 parameter of the *getsockopt()* or *setsockopt()* function is SOL_SOCKET, the types of the option
 2587 value parameters associated with each option, and a brief synopsis of the meaning of the option
 2588 value parameter. Unless otherwise noted, each may be examined with *getsockopt()* and set with
 2589 *setsockopt()* on all types of socket.

Table 2-2 Socket-Level Options

Option	Parameter Type	Parameter Meaning
SO_BROADCAST	int	Non-zero requests permission to transmit broadcast datagrams (SOCK_DGRAM sockets only).
SO_DEBUG	int	Non-zero requests debugging in underlying protocol modules.
SO_DONTROUTE	int	Non-zero requests bypass of normal routing; route based on destination address only.
SO_ERROR	int	Requests and clears pending error information on the socket (<i>getsockopt()</i> only).
SO_KEEPALIVE	int	Non-zero requests periodic transmission of keepalive messages (protocol-specific).
SO_LINGER	struct linger	Specify actions to be taken for queued, unsent data on <i>close()</i> : linger on/off and linger time in seconds.
SO_OOBINLINE	int	Non-zero requests that out-of-band data be placed into normal data input queue as received.
SO_RCVBUF	int	Size of receive buffer (in bytes).
SO_RCVLOWAT	int	Minimum amount of data to return to application for input operations (in bytes).
SO_RCVTIMEO	struct timeval	Timeout value for a socket receive operation.
SO_REUSEADDR	int	Non-zero requests reuse of local addresses in <i>bind()</i> (protocol-specific).
SO_SNDBUF	int	Size of send buffer (in bytes).
SO_SNDLOWAT	int	Minimum amount of data to send for output operations (in bytes).
SO_SNDTIMEO	struct timeval	Timeout value for a socket send operation.
SO_TYPE	int	Identify socket type (<i>getsockopt()</i> only).

The SO_BROADCAST option requests permission to send broadcast datagrams on the socket. Support for SO_BROADCAST is protocol-specific. The default for SO_BROADCAST is that the ability to send broadcast datagrams on a socket is disabled.

The SO_DEBUG option enables debugging in the underlying protocol modules. This can be useful for tracing the behavior of the underlying protocol modules during normal system operation. The semantics of the debug reports are implementation-defined. The default value for SO_DEBUG is for debugging to be turned off.

The SO_DONTROUTE option requests that outgoing messages bypass the standard routing facilities. The destination must be on a directly-connected network, and messages are directed to the appropriate network interface according to the destination address. It is protocol-specific whether this option has any effect and how the outgoing network interface is chosen. Support for this option with each protocol is implementation-defined.

The SO_ERROR option is used only on *getsockopt()*. When this option is specified, *getsockopt()* shall return any pending error on the socket and clear the error status. It shall return a value of 0 if there is no pending error. SO_ERROR may be used to check for asynchronous errors on connected connectionless-mode sockets or for other types of asynchronous errors. SO_ERROR has no default value.

2637 The SO_KEEPALIVE option enables the periodic transmission of messages on a connected
2638 socket. The behavior of this option is protocol-specific. The default value for SO_KEEPALIVE is
2639 zero, specifying that this capability is turned off.

2640 The SO_LINGER option controls the action of the interface when unsent messages are queued
2641 on a socket and a *close()* is performed. The details of this option are protocol-specific. The
2642 default value for SO_LINGER is zero, or off, for the *L_onoff* element of the option value and zero
2643 seconds for the linger time specified by the *L_linger* element.

2644 The SO_OOINLINE option is valid only on protocols that support out-of-band data. The
2645 SO_OOINLINE option requests that out-of-band data be placed in the normal data input queue
2646 as received; it is then accessible using the *read()* or *recv()* functions without the MSG_OOB flag
2647 set. The default for SO_OOINLINE is off; that is, for out-of-band data not to be placed in the
2648 normal data input queue.

2649 The SO_RCVBUF option requests that the buffer space allocated for receive operations on this
2650 socket be set to the value, in bytes, of the option value. Applications may wish to increase buffer
2651 size for high volume connections, or may decrease buffer size to limit the possible backlog of
2652 incoming data. The default value for the SO_RCVBUF option value is implementation-defined,
2653 and may vary by protocol.

2654 The maximum value for the option for a socket may be obtained by the use of the *fpathconf()*
2655 function, using the value *_PC_SOCKET_MAXBUF*.

2656 The SO_RCVLOWAT option sets the minimum number of bytes to process for socket input
2657 operations. In general, receive calls block until any (non-zero) amount of data is received, then
2658 return the smaller of the amount available or the amount requested. The default value for
2659 SO_RCVLOWAT is 1, and does not affect the general case. If SO_RCVLOWAT is set to a larger
2660 value, blocking receive calls normally wait until they have received the smaller of the low water
2661 mark value or the requested amount. Receive calls may still return less than the low water mark
2662 if an error occurs, a signal is caught, or the type of data next in the receive queue is different
2663 from that returned (for example, out-of-band data). As mentioned previously, the default value
2664 for SO_RCVLOWAT is 1 byte. It is implementation-defined whether the SO_RCVLOWAT option
2665 can be set.

2666 The SO_RCVTIMEO option is an option to set a timeout value for input operations. It accepts a
2667 **timeval** structure with the number of seconds and microseconds specifying the limit on how
2668 long to wait for an input operation to complete. If a receive operation has blocked for this much
2669 time without receiving additional data, it shall return with a partial count or *errno* shall be set to
2670 [EWOULDBLOCK] if no data were received. The default for this option is the value zero, which
2671 indicates that a receive operation will not time out. It is implementation-defined whether the
2672 SO_RCVTIMEO option can be set.

2673 The SO_REUSEADDR option indicates that the rules used in validating addresses supplied in a
2674 *bind()* should allow reuse of local addresses. Operation of this option is protocol-specific. The
2675 default value for SO_REUSEADDR is off; that is, reuse of local addresses is not permitted.

2676 The SO_SNDBUF option requests that the buffer space allocated for send operations on this
2677 socket be set to the value, in bytes, of the option value. The default value for the SO_SNDBUF
2678 option value is implementation-defined, and may vary by protocol. The maximum value for the
2679 option for a socket may be obtained by the use of the *fpathconf()* function, using the value
2680 *_PC_SOCKET_MAXBUF*.

2681 The SO_SNDLOWAT option sets the minimum number of bytes to process for socket output
2682 operations. Most output operations process all of the data supplied by the call, delivering data to
2683 the protocol for transmission and blocking as necessary for flow control. Non-blocking output
2684 operations process as much data as permitted subject to flow control without blocking, but

process no data if flow control does not allow the smaller of the send low water mark value or the entire request to be processed. A *select()* operation testing the ability to write to a socket shall return true only if the send low water mark could be processed. The default value for `SO_SNDLOWAT` is implementation-defined and protocol-specific. It is implementation-defined whether the `SO_SNDLOWAT` option can be set.

The `SO_SNDTIMEO` option is an option to set a timeout value for the amount of time that an output function shall block because flow control prevents data from being sent. As noted in Table 2-2 (on page 64), the option value is a **timeval** structure with the number of seconds and microseconds specifying the limit on how long to wait for an output operation to complete. If a send operation has blocked for this much time, it shall return with a partial count or *errno* set to `[EWOULDBLOCK]` if no data were sent. The default for this option is the value zero, which indicates that a send operation will not time out. It is implementation-defined whether the `SO_SNDTIMEO` option can be set.

The `SO_TYPE` option is used only on *getsockopt()*. When this option is specified, *getsockopt()* shall return the type of the socket (for example, `SOCK_STREAM`). This option is useful to servers that inherit sockets on start-up. `SO_TYPE` has no default value.

2.10.17 Use of Sockets for Local UNIX Connections

Support for UNIX domain sockets is mandatory.

UNIX domain sockets provide process-to-process communication in a single system.

2.10.17.1 Headers

The symbolic constant `AF_UNIX` defined in the `<sys/socket.h>` header is used to identify the UNIX domain address family. The `<sys/un.h>` header contains other definitions used in connection with UNIX domain sockets. See the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 13, Headers.

The **sockaddr_storage** structure defined in `<sys/socket.h>` shall be large enough to accommodate a **sockaddr_un** structure (see the `<sys/un.h>` header defined in the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 13, Headers) and shall be aligned at an appropriate boundary so that pointers to it can be cast as pointers to **sockaddr_un** structures and used to access the fields of those structures without alignment problems. When a **sockaddr_storage** structure is cast as a **sockaddr_un** structure, the *ss_family* field maps onto the *sun_family* field.

2.10.18 Use of Sockets over Internet Protocols

When a socket is created in the Internet family with a protocol value of zero, the implementation shall use the protocol listed below for the type of socket created.

`SOCK_STREAM` `IPPROTO_TCP`.

`SOCK_DGRAM` `IPPROTO_UDP`.

RS `SOCK_RAW` `IPPROTO_RAW`.

`SOCK_SEQPACKET` Unspecified.

RS A raw interface to IP is available by creating an Internet socket of type `SOCK_RAW`. The default protocol for type `SOCK_RAW` shall be identified in the IP header with the value `IPPROTO_RAW`. Applications should not use the default protocol when creating a socket with type `SOCK_RAW`, but should identify a specific protocol by value. The ICMP control protocol is accessible from a raw socket by specifying a value of `IPPROTO_ICMP` for protocol.

2.10.19 Use of Sockets over Internet Protocols Based on IPv4

Support for sockets over Internet protocols based on IPv4 is mandatory.

2.10.19.1 Headers

The symbolic constant `AF_INET` defined in the `<sys/socket.h>` header is used to identify the IPv4 Internet address family. The `<netinet/in.h>` header contains other definitions used in connection with IPv4 Internet sockets. See the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 13, Headers.

The `sockaddr_storage` structure defined in `<sys/socket.h>` shall be large enough to accommodate a `sockaddr_in` structure (see the `<netinet/in.h>` header defined in the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 13, Headers) and shall be aligned at an appropriate boundary so that pointers to it can be cast as pointers to `sockaddr_in` structures and used to access the fields of those structures without alignment problems. When a `sockaddr_storage` structure is cast as a `sockaddr_in` structure, the `ss_family` field maps onto the `sin_family` field.

2.10.20 Use of Sockets over Internet Protocols Based on IPv6

This section describes extensions to support sockets over Internet protocols based on IPv6. This functionality is dependent on support of the IPV6 option (and the rest of this section is not further shaded for this option).

To enable smooth transition from IPv4 to IPv6, the features defined in this section may, in certain circumstances, also be used in connection with IPv4; see Section 2.10.20.2 (on page 68).

2.10.20.1 Addressing

IPv6 overcomes the addressing limitations of previous versions by using 128-bit addresses instead of 32-bit addresses. The IPv6 address architecture is described in RFC 2373.

There are three kinds of IPv6 address:

Unicast

Identifies a single interface.

A unicast address can be global, link-local (designed for use on a single link), or site-local (designed for systems not connected to the Internet). Link-local and site-local addresses need not be globally unique.

Anycast

Identifies a set of interfaces such that a packet sent to the address can be delivered to any member of the set.

An anycast address is similar to a unicast address; the nodes to which an anycast address is assigned must be explicitly configured to know that it is an anycast address.

Multicast

Identifies a set of interfaces such that a packet sent to the address should be delivered to every member of the set.

An application can send multicast datagrams by simply specifying an IPv6 multicast address in the `address` argument of `sendto()`. To receive multicast datagrams, an application must join the multicast group (using `setsockopt()` with `IPV6_JOIN_GROUP`) and must bind to the socket the UDP port on which datagrams will be received. Some applications should also bind the multicast group address to the socket, to prevent other datagrams destined to that port from being delivered to the socket.

2771 A multicast address can be global, node-local, link-local, site-local, or organization-local.

2772 The following special IPv6 addresses are defined:

2773 Unspecified

2774 An address that is not assigned to any interface and is used to indicate the absence of an

2775 address.

2776 Loopback

2777 A unicast address that is not assigned to any interface and can be used by a node to send

2778 packets to itself.

2779 Two sets of IPv6 addresses are defined to correspond to IPv4 addresses:

2780 IPv4-compatible addresses

2781 These are assigned to nodes that support IPv6 and can be used when traffic is “tunneled”

2782 through IPv4.

2783 IPv4-mapped addresses

2784 These are used to represent IPv4 addresses in IPv6 address format; see Section 2.10.20.2.

2785 Note that the unspecified address and the loopback address must not be treated as IPv4-

2786 compatible addresses.

2787 2.10.20.2 Compatibility with IPv4

2788 The API provides the ability for IPv6 applications to interoperate with applications using IPv4,

2789 by using IPv4-mapped IPv6 addresses. These addresses can be generated automatically by the

2790 `getaddrinfo()` function when the specified host has only IPv4 addresses.

2791 Applications can use `AF_INET6` sockets to open TCP connections to IPv4 nodes, or send UDP

2792 packets to IPv4 nodes, by simply encoding the destination's IPv4 address as an IPv4-mapped

2793 IPv6 address, and passing that address, within a `sockaddr_in6` structure, in the `connect()`,

2794 `sendto()`, or `sendmsg()` function. When applications use `AF_INET6` sockets to accept TCP

2795 connections from IPv4 nodes, or receive UDP packets from IPv4 nodes, the system shall return

2796 the peer's address to the application in the `accept()`, `recvfrom()`, `recvmsg()`, or `getpeername()`

2797 function using a `sockaddr_in6` structure encoded this way. If a node has an IPv4 address, then

2798 the implementation shall allow applications to communicate using that address via an

2799 `AF_INET6` socket. In such a case, the address will be represented at the API by the

2800 corresponding IPv4-mapped IPv6 address. Also, the implementation may allow an `AF_INET6`

2801 socket bound to `in6addr_any` to receive inbound connections and packets destined to one of the

2802 node's IPv4 addresses.

2803 An application can use `AF_INET6` sockets to bind to a node's IPv4 address by specifying the

2804 address as an IPv4-mapped IPv6 address in a `sockaddr_in6` structure in the `bind()` function. For

2805 an `AF_INET6` socket bound to a node's IPv4 address, the system shall return the address in the

2806 `getsockname()` function as an IPv4-mapped IPv6 address in a `sockaddr_in6` structure.

2807 2.10.20.3 Interface Identification

2808 Each local interface is assigned a unique positive integer as a numeric index. Indexes start at 1;

2809 zero is not used. There may be gaps so that there is no current interface for a particular positive

2810 index. Each interface also has a unique implementation-defined name.

2811 2.10.20.4 Options

2812 The following options apply at the IPPROTO_IPV6 level:

2813 IPV6_JOIN_GROUP

2814 When set via *setsockopt()*, it joins the application to a multicast group on an interface
 2815 (identified by its index) and addressed by a given multicast address, enabling packets sent
 2816 to that address to be read via the socket. If the interface index is specified as zero, the
 2817 system selects the interface (for example, by looking up the address in a routing table and
 2818 using the resulting interface).

2819 An attempt to read this option using *getsockopt()* shall result in an [EOPNOTSUPP] error.

2820 The parameter type of this option is a pointer to an **ipv6_mreq** structure.

2821 IPV6_LEAVE_GROUP

2822 When set via *setsockopt()*, it removes the application from the multicast group on an
 2823 interface (identified by its index) and addressed by a given multicast address.

2824 An attempt to read this option using *getsockopt()* shall result in an [EOPNOTSUPP] error.

2825 The parameter type of this option is a pointer to an **ipv6_mreq** structure.

2826 IPV6_MULTICAST_HOPS

2827 The value of this option is the hop limit for outgoing multicast IPv6 packets sent via the
 2828 socket. Its possible values are the same as those of IPV6_UNICAST_HOPS. If the
 2829 IPV6_MULTICAST_HOPS option is not set, a value of 1 is assumed. This option can be set
 2830 via *setsockopt()* and read via *getsockopt()*.

2831 The parameter type of this option is a pointer to an **int**. (Default value: 1)

2832 IPV6_MULTICAST_IF

2833 The index of the interface to be used for outgoing multicast packets. It can be set via
 2834 *setsockopt()* and read via *getsockopt()*. If the interface index is specified as zero, the system
 2835 selects the interface (for example, by looking up the address in a routing table and using the
 2836 resulting interface).

2837 The parameter type of this option is a pointer to an **unsigned int**. (Default value: 0)

2838 IPV6_MULTICAST_LOOP

2839 This option controls whether outgoing multicast packets should be delivered back to the
 2840 local application when the sending interface is itself a member of the destination multicast
 2841 group. If it is set to 1 they are delivered. If it is set to 0 they are not. Other values result in an
 2842 [EINVAL] error. This option can be set via *setsockopt()* and read via *getsockopt()*.

2843 The parameter type of this option is a pointer to an **unsigned int** which is used as a Boolean
 2844 value. (Default value: 1)

2845 IPV6_UNICAST_HOPS

2846 The value of this option is the hop limit for outgoing unicast IPv6 packets sent via the
 2847 socket. If the option is not set, or is set to -1, the system selects a default value. Attempts to
 2848 set a value less than -1 or greater than 255 shall result in an [EINVAL] error. This option can
 2849 be set via *setsockopt()* and read via *getsockopt()*.

2850 The parameter type of this option is a pointer to an **int**. (Default value: Unspecified)

2851 IPV6_V6ONLY

2852 This socket option restricts AF_INET6 sockets to IPv6 communications only. AF_INET6
 2853 sockets may be used for both IPv4 and IPv6 communications. Some applications may want
 2854 to restrict their use of an AF_INET6 socket to IPv6 communications only. For these

2855 applications, the IPv6_V6ONLY socket option is defined. When this option is turned on, the
 2856 socket can be used to send and receive IPv6 packets only. This is an IPPROTO_IPV6-level
 2857 option.

2858 The parameter type of this option is a pointer to an **int** which is used as a Boolean value.
 2859 (Default value: 0)

2860 An [EOPNOTSUPP] error shall result if IPV6_JOIN_GROUP or IPV6_LEAVE_GROUP is used
 2861 with *getsockopt()*.

2862 2.10.20.5 Headers

2863 The symbolic constant AF_INET6 is defined in the **<sys/socket.h>** header to identify the IPv6
 2864 Internet address family. See the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 13,
 2865 Headers.

2866 The **sockaddr_storage** structure defined in **<sys/socket.h>** shall be large enough to
 2867 accommodate a **sockaddr_in6** structure (see the **<netinet/in.h>** header defined in the Base
 2868 Definitions volume of IEEE Std 1003.1-2001, Chapter 13, Headers) and shall be aligned at an
 2869 appropriate boundary so that pointers to it can be cast as pointers to **sockaddr_in6** structures
 2870 and used to access the fields of those structures without alignment problems. When a
 2871 **sockaddr_storage** structure is cast as a **sockaddr_in6** structure, the *ss_family* field maps onto the
 2872 *sin6_family* field.

2873 The **<netinet/in.h>**, **<arpa/inet.h>**, and **<netdb.h>** headers contain other definitions used in
 2874 connection with IPv6 Internet sockets; see the Base Definitions volume of IEEE Std 1003.1-2001,
 2875 Chapter 13, Headers.

2876 2.11 Tracing

2877 TRC This section describes extensions to support tracing of user applications. This functionality is
 2878 dependent on support of the Trace option (and the rest of this section is not further shaded for
 2879 this option).

2880 The tracing facilities defined in IEEE Std 1003.1-2001 allow a process to select a set of trace event
 2881 types, to activate a trace stream of the selected trace events as they occur in the flow of
 2882 execution, and to retrieve the recorded trace events.

2883 The tracing operation relies on three logically different components: the traced process, the
 2884 controller process, and the analyzer process. During the execution of the traced process, when a
 2885 trace point is reached, a trace event is recorded into the trace streams created for that process in
 2886 which the associated trace event type identifier is not being filtered out. The controller process
 2887 controls the operation of recording the trace events into the trace stream. It shall be able to:

- 2888 • Initialize the attributes of a trace stream
- 2889 • Create the trace stream (for a specified traced process) using those attributes
- 2890 • Start and stop tracing for the trace stream
- 2891 • Filter the type of trace events to be recorded, if the Trace Event Filter option is supported
- 2892 • Shut a trace stream down

2893 These operations can be done for an active trace stream. The analyzer process retrieves the
 2894 traced events either at runtime, when the trace stream has not yet been shut down, but is still
 2895 recording trace events; or after opening a trace log that had been previously recorded and shut
 2896 down. These three logically different operations can be performed by the same process, or can be

distributed into different processes.

A trace stream identifier can be created by a call to `posix_trace_create()`, `posix_trace_create_withlog()`, or `posix_trace_open()`. The `posix_trace_create()` and `posix_trace_create_withlog()` functions should be used by a controller process. The `posix_trace_open()` should be used by an analyzer process.

The tracing functions can serve different purposes. One purpose is debugging the possibly pre-instrumented code, while another is post-mortem fault analysis. These two potential uses differ in that the first requires pre-filtering capabilities to avoid overwhelming the trace stream and permits focusing on expected information; while the second needs comprehensive trace capabilities in order to be able to record all types of information.

The events to be traced belong to two classes:

1. User trace events (generated by the application instrumentation)
2. System trace events (generated by the operating system)

The trace interface defines several system trace event types associated with control of and operation of the trace stream. This small set of system trace events includes the minimum required to interpret correctly the trace event information present in the stream. Other desirable system trace events for some particular application profile may be implemented and are encouraged; for example, process and thread scheduling, signal occurrence, and so on.

Each traced process shall have a mapping of the trace event names to trace event type identifiers that have been defined for that process. Each active trace stream shall have a mapping that incorporates all the trace event type identifiers predefined by the trace system plus all the mappings of trace event names to trace event type identifiers of the processes that are being traced into that trace stream. These mappings are defined from the instrumented application by calling the `posix_trace_eventid_open()` function and from the controller process by calling the `posix_trace_trid_eventid_open()` function. For a pre-recorded trace stream, the list of trace event types is obtained from the pre-recorded trace log.

The `st_ctime` and `st_mtime` fields of a file associated with an active trace stream shall be marked for update every time any of the tracing operations modifies that file.

The `st_atime` field of a file associated with a trace stream shall be marked for update every time any of the tracing operations causes data to be read from that file.

Results are undefined if the application performs any operation on a file descriptor associated with an active or pre-recorded trace stream until `posix_trace_shutdown()` or `posix_trace_close()` is called for that trace stream.

The main purpose of this option is to define a complete set of functions and concepts that allow a conforming application to be traced from creation to termination, whatever its realtime constraints and properties.

2.11.1 Tracing Data Definitions

2.11.1.1 Structures

The `<trace.h>` header shall define the `posix_trace_status_info` and `posix_trace_event_info` structures described below. Implementations may add extensions to these structures.

posix_trace_status_info Structure

To facilitate control of a trace stream, information about the current state of an active trace stream can be obtained dynamically. This structure is returned by a call to the *posix_trace_get_status()* function.

The **posix_trace_status_info** structure defined in **<trace.h>** shall contain at least the following members:

Member Type	Member Name	Description
int	<i>posix_stream_status</i>	The operating mode of the trace stream.
int	<i>posix_stream_full_status</i>	The full status of the trace stream.
int	<i>posix_stream_overnrun_status</i>	Indicates whether trace events were lost in the trace stream.

If the Trace Log option is supported in addition to the Trace option, the **posix_trace_status_info** structure defined in **<trace.h>** shall contain at least the following additional members:

Member Type	Member Name	Description
int	<i>posix_stream_flush_status</i>	Indicates whether a flush is in progress.
int	<i>posix_stream_flush_error</i>	Indicates whether any error occurred during the last flush operation.
int	<i>posix_log_overnrun_status</i>	Indicates whether trace events were lost in the trace log.
int	<i>posix_log_full_status</i>	The full status of the trace log.

The *posix_stream_status* member indicates the operating mode of the trace stream and shall have one of the following values defined by manifest constants in the **<trace.h>** header:

POSIX_TRACE_RUNNING

Tracing is in progress; that is, the trace stream is accepting trace events.

POSIX_TRACE_SUSPENDED

The trace stream is not accepting trace events. The tracing operation has not yet started or has stopped, either following a *posix_trace_stop()* function call or because the trace resources are exhausted.

The *posix_stream_full_status* member indicates the full status of the trace stream, and it shall have one of the following values defined by manifest constants in the **<trace.h>** header:

POSIX_TRACE_FULL

The space in the trace stream for trace events is exhausted.

POSIX_TRACE_NOT_FULL

There is still space available in the trace stream.

The combination of the *posix_stream_status* and *posix_stream_full_status* members also indicates the actual status of the stream. The status shall be interpreted as follows:

POSIX_TRACE_RUNNING and POSIX_TRACE_NOT_FULL

This status combination indicates that tracing is in progress, and there is space available for recording more trace events.

POSIX_TRACE_RUNNING and POSIX_TRACE_FULL

This status combination indicates that tracing is in progress and that the trace stream is full of trace events. This status combination cannot occur unless the *stream-full-policy* is set to

2981 POSIX_TRACE_LOOP. The trace stream contains trace events recorded during a moving
 2982 time window of prior trace events, and some older trace events may have been overwritten
 2983 and thus lost.

2984 POSIX_TRACE_SUSPENDED and POSIX_TRACE_NOT_FULL
 2985 This status combination indicates that tracing has not yet been started, has been stopped by
 2986 the *posix_trace_stop()* function, or has been cleared by the *posix_trace_clear()* function.

2987 POSIX_TRACE_SUSPENDED and POSIX_TRACE_FULL
 2988 This status combination indicates that tracing has been stopped by the implementation
 2989 because the *stream-full-policy* attribute was POSIX_TRACE_UNTIL_FULL and trace
 2990 resources were exhausted, or that the trace stream was stopped by the function
 2991 *posix_trace_stop()* at a time when trace resources were exhausted.

2992 The *posix_stream_outrun_status* member indicates whether trace events were lost in the trace
 2993 stream, and shall have one of the following values defined by manifest constants in the
 2994 **<trace.h>** header:

2995 POSIX_TRACE_OVERRUN
 2996 At least one trace event was lost and thus was not recorded in the trace stream.

2997 POSIX_TRACE_NO_OVERRUN
 2998 No trace events were lost.

2999 When the corresponding trace stream is created, the *posix_stream_outrun_status* member shall be
 3000 set to POSIX_TRACE_NO_OVERRUN.

3001 Whenever an overrun occurs, the *posix_stream_outrun_status* member shall be set to
 3002 POSIX_TRACE_OVERRUN.

3003 An overrun occurs when:

- 3004 • The policy is POSIX_TRACE_LOOP and a recorded trace event is overwritten.
- 3005 • The policy is POSIX_TRACE_UNTIL_FULL and the trace stream is full when a trace event is
 3006 generated.
- 3007 • If the Trace Log option is supported, the policy is POSIX_TRACE_FLUSH and at least one
 3008 trace event is lost while flushing the trace stream to the trace log.

3009 The *posix_stream_outrun_status* member is reset to zero after its value is read.

3010 If the Trace Log option is supported in addition to the Trace option, the *posix_stream_flush_status*,
 3011 *posix_stream_flush_error*, *posix_log_outrun_status*, and *posix_log_full_status* members are defined
 3012 as follows; otherwise, they are undefined.

3013 The *posix_stream_flush_status* member indicates whether a flush operation is being performed
 3014 and shall have one of the following values defined by manifest constants in the header
 3015 **<trace.h>**:

3016 POSIX_TRACE_FLUSHING
 3017 The trace stream is currently being flushed to the trace log.

3018 POSIX_TRACE_NOT_FLUSHING
 3019 No flush operation is in progress.

3020 The *posix_stream_flush_status* member shall be set to POSIX_TRACE_FLUSHING if a flush
 3021 operation is in progress either due to a call to the *posix_trace_flush()* function (explicit or caused
 3022 by a trace stream shutdown operation) or because the trace stream has become full with the
 3023 *stream-full-policy* attribute set to POSIX_TRACE_FLUSH. The *posix_stream_flush_status* member
 3024 shall be set to POSIX_TRACE_NOT_FLUSHING if no flush operation is in progress.

The *posix_stream_flush_error* member shall be set to zero if no error occurred during flushing. If an error occurred during a previous flushing operation, the *posix_stream_flush_error* member shall be set to the value of the first error that occurred. If more than one error occurs while flushing, error values after the first shall be discarded. The *posix_stream_flush_error* member is reset to zero after its value is read.

The *posix_log_overrun_status* member indicates whether trace events were lost in the trace log, and shall have one of the following values defined by manifest constants in the `<trace.h>` header:

POSIX_TRACE_OVERRUN

At least one trace event was lost.

POSIX_TRACE_NO_OVERRUN

No trace events were lost.

When the corresponding trace stream is created, the *posix_log_overrun_status* member shall be set to POSIX_TRACE_NO_OVERRUN. Whenever an overrun occurs, this status shall be set to POSIX_TRACE_OVERRUN. The *posix_log_overrun_status* member is reset to zero after its value is read.

The *posix_log_full_status* member indicates the full status of the trace log, and it shall have one of the following values defined by manifest constants in the `<trace.h>` header:

POSIX_TRACE_FULL

The space in the trace log is exhausted.

POSIX_TRACE_NOT_FULL

There is still space available in the trace log.

The *posix_log_full_status* member is only meaningful if the *log-full-policy* attribute is either POSIX_TRACE_UNTIL_FULL or POSIX_TRACE_LOOP.

For an active trace stream without log, that is created by the *posix_trace_create()* function, the *posix_log_overrun_status* member shall be set to POSIX_TRACE_NO_OVERRUN and the *posix_log_full_status* member shall be set to POSIX_TRACE_NOT_FULL.

posix_trace_event_info Structure

The trace event structure **posix_trace_event_info** contains the information for one recorded trace event. This structure is returned by the set of functions *posix_trace_getnext_event()*, *posix_trace_timedgetnext_event()*, and *posix_trace_trygetnext_event()*.

The **posix_trace_event_info** structure defined in `<trace.h>` shall contain at least the following members:

Member Type	Member Name	Description
trace_event_id_t pid_t	<i>posix_event_id</i> <i>posix_pid</i>	Trace event type identification. Process ID of the process that generated the trace event.
void * int	<i>posix_prog_address</i> <i>posix_truncation_status</i>	Address at which the trace point was invoked. Status about the truncation of the data associated with this trace event.
struct timespec	<i>posix_timestamp</i>	Time at which the trace event was generated.

In addition, if the Trace option and the Threads option are both supported, the **posix_trace_event_info** structure defined in `<trace.h>` shall contain the following additional member:

3069
3070
3071
3072
3073

Member Type	Member Name	Description
pthread_t	<i>posix_thread_id</i>	Thread ID of the thread that generated the trace event.

3074
3075
3076
3077
3078

The *posix_event_id* member represents the identification of the trace event type and its value is not directly defined by the user. This identification is returned by a call to one of the following functions: *posix_trace_trid_eventid_open()*, *posix_trace_eventtypelist_getnext_id()*, or *posix_trace_eventid_open()*. The name of the trace event type can be obtained by calling *posix_trace_eventid_get_name()*.

3079
3080
3081

The *posix_pid* is the process identifier of the traced process which generated the trace event. If the *posix_event_id* member is one of the implementation-defined system trace events and that trace event is not associated with any process, the *posix_pid* member shall be set to zero.

3082
3083
3084
3085
3086

For a user trace event, the *posix_prog_address* member is the process mapped address of the point at which the associated call to the *posix_trace_event()* function was made. For a system trace event, if the trace event is caused by a system service explicitly called by the application, the *posix_prog_address* member shall be the address of the process at the point where the call to that system service was made.

3087
3088
3089
3090
3091

The *posix_truncation_status* member defines whether the data associated with a trace event has been truncated at the time the trace event was generated, or at the time the trace event was read from the trace stream, or (if the Trace Log option is supported) from the trace log (see the *event* argument from the *posix_trace_getnext_event()* function). The *posix_truncation_status* member shall have one of the following values defined by manifest constants in the **<trace.h>** header:

3092
3093

POSIX_TRACE_NOT_TRUNCATED

All the traced data is available.

3094
3095

POSIX_TRACE_TRUNCATED_RECORD

Data was truncated at the time the trace event was generated.

3096
3097
3098
3099

POSIX_TRACE_TRUNCATED_READ

Data was truncated at the time the trace event was read from a trace stream or a trace log because the reader's buffer was too small. This truncation status overrides the POSIX_TRACE_TRUNCATED_RECORD status.

3100
3101
3102

The *posix_timestamp* member shall be the time at which the trace event was generated. The clock used is implementation-defined, but the resolution of this clock can be retrieved by a call to the *posix_trace_attr_getclockres()* function.

3103

If the Threads option is supported in addition to the Trace option:

3104
3105
3106

- The *posix_thread_id* member is the identifier of the thread that generated the trace event. If the *posix_event_id* member is one of the implementation-defined system trace events and that trace event is not associated with any thread, the *posix_thread_id* member shall be set to zero.

3107

Otherwise, this member is undefined.

3108 2.11.1.2 Trace Stream Attributes

3109
3110

Trace streams have attributes that compose the **posix_trace_attr_t** trace stream attributes object. This object shall contain at least the following attributes:

3111

- The *generation-version* attribute identifies the origin and version of the trace system.

- 3112 • The *trace-name* attribute is a character string defined by the trace controller, and that
3113 identifies the trace stream.
- 3114 • The *creation-time* attribute represents the time of the creation of the trace stream.
- 3115 • The *clock-resolution* attribute defines the clock resolution of the clock used to generate
3116 timestamps.
- 3117 • The *stream-min-size* attribute defines the minimum size in bytes of the trace stream strictly
3118 reserved for the trace events.
- 3119 • The *stream-full-policy* attribute defines the policy followed when the trace stream is full; its
3120 value is `POSIX_TRACE_LOOP`, `POSIX_TRACE_UNTIL_FULL`, or `POSIX_TRACE_FLUSH`.
- 3121 • The *max-data-size* attribute defines the maximum record size in bytes of a trace event.

3122 In addition, if the Trace option and the Trace Inherit option are both supported, the
3123 **posix_trace_attr_t** trace stream creation attributes object shall contain at least the following
3124 attributes:

- 3125 • The *inheritance* attribute specifies whether a newly created trace stream will inherit tracing in
3126 its parent's process trace stream. It is either `POSIX_TRACE_INHERITED` or
3127 `POSIX_TRACE_CLOSE_FOR_CHILD`.

3128 In addition, if the Trace option and the Trace Log option are both supported, the
3129 **posix_trace_attr_t** trace stream creation attributes object shall contain at least the following
3130 attribute:

- 3131 • If the file type corresponding to the trace log supports the `POSIX_TRACE_LOOP` or the
3132 `POSIX_TRACE_UNTIL_FULL` policies, the *log-max-size* attribute defines the maximum size
3133 in bytes of the trace log associated with an active trace stream. Other stream data—for
3134 example, trace attribute values—shall not be included in this size.
- 3135 • The *log-full-policy* attribute defines the policy of a trace log associated with an active trace
3136 stream to be `POSIX_TRACE_LOOP`, `POSIX_TRACE_UNTIL_FULL`, or
3137 `POSIX_TRACE_APPEND`.

3138 2.11.2 Trace Event Type Definitions

3139 2.11.2.1 System Trace Event Type Definitions

3140 The following system trace event types, defined in the `<trace.h>` header, track the invocation of
3141 the trace operations:

- 3142 • `POSIX_TRACE_START` shall be associated with a trace start operation.
- 3143 • `POSIX_TRACE_STOP` shall be associated with a trace stop operation.
- 3144 • If the Trace Event Filter option is supported, `POSIX_TRACE_FILTER` shall be associated with
3145 a trace event type filter change operation.

3146 The following system trace event types, defined in the `<trace.h>` header, report operational trace
3147 events:

- 3148 • `POSIX_TRACE_OVERFLOW` shall mark the beginning of a trace overflow condition.
- 3149 • `POSIX_TRACE_RESUME` shall mark the end of a trace overflow condition.
- 3150 • If the Trace Log option is supported, `POSIX_TRACE_FLUSH_START` shall mark the
3151 beginning of a flush operation.

- If the Trace Log option is supported, POSIX_TRACE_FLUSH_STOP shall mark the end of a flush operation.

- If an implementation-defined trace error condition is reported, it shall be marked POSIX_TRACE_ERROR.

The interpretation of a trace stream or a trace log by a trace analyzer process relies on the information recorded for each trace event, and also on system trace events that indicate the invocation of trace control operations and trace system operational trace events.

The POSIX_TRACE_START and POSIX_TRACE_STOP trace events specify the time windows during which the trace stream is running.

- The POSIX_TRACE_STOP trace event with an associated data that is equal to zero indicates a call of the function *posix_trace_stop()*.
- The POSIX_TRACE_STOP trace event with an associated data that is different from zero indicates an automatic stop of the trace stream (see *posix_trace_attr_getstreamfullpolicy()*).

The POSIX_TRACE_FILTER trace event indicates that a trace event type filter value changed while the trace stream was running.

The POSIX_TRACE_ERROR serves to inform the analyzer process that an implementation-defined internal error of the trace system occurred.

The POSIX_TRACE_OVERFLOW trace event shall be reported with a timestamp equal to the timestamp of the first trace event overwritten. This is an indication that some generated trace events have been lost.

The POSIX_TRACE_RESUME trace event shall be reported with a timestamp equal to the timestamp of the first valid trace event reported after the overflow condition ends and shall be reported before this first valid trace event. This is an indication that the trace system is reliably recording trace events after an overflow condition.

Each of these trace event types shall be defined by a constant trace event name and a **trace_event_id_t** constant; trace event data is associated with some of these trace events.

If the Trace option is supported and the Trace Event Filter option and the Trace Log option are not supported, the following predefined system trace events in Table 2-3 shall be defined:

Table 2-3 Trace Option: System Trace Events

Event Name	Constant	Associated Data
		Data Type
posix_trace_error	POSIX_TRACE_ERROR	error
		int
posix_trace_start	POSIX_TRACE_START	None.
posix_trace_stop	POSIX_TRACE_STOP	auto
		int
posix_trace_overflow	POSIX_TRACE_OVERFLOW	None.
posix_trace_resume	POSIX_TRACE_RESUME	None.

If the Trace option and the Trace Event Filter option are both supported, and if the Trace Log option is not supported, the following predefined system trace events in Table 2-4 (on page 78) shall be defined:

Table 2-4 Trace and Trace Event Filter Options: System Trace Events

Event Name	Constant	Associated Data
		Data Type
posix_trace_error	POSIX_TRACE_ERROR	error int
posix_trace_start	POSIX_TRACE_START	event_filter trace_event_set_t
posix_trace_stop	POSIX_TRACE_STOP	auto int
posix_trace_filter	POSIX_TRACE_FILTER	old_event_filter new_event_filter trace_event_set_t
posix_trace_overflow	POSIX_TRACE_OVERFLOW	None.
posix_trace_resume	POSIX_TRACE_RESUME	None.

If the Trace option and the Trace Log option are both supported, and if the Trace Event Filter option is not supported, the following predefined system trace events in Table 2-5 shall be defined:

Table 2-5 Trace and Trace Log Options: System Trace Events

Event Name	Constant	Associated Data
		Data Type
posix_trace_error	POSIX_TRACE_ERROR	error int
posix_trace_start	POSIX_TRACE_START	None.
posix_trace_stop	POSIX_TRACE_STOP	auto int
posix_trace_overflow	POSIX_TRACE_OVERFLOW	None.
posix_trace_resume	POSIX_TRACE_RESUME	None.
posix_trace_flush_start	POSIX_TRACE_FLUSH_START	None.
posix_trace_flush_stop	POSIX_TRACE_FLUSH_STOP	None.

If the Trace option, the Trace Event Filter option, and the Trace Log option are all supported, the following predefined system trace events in Table 2-6 (on page 79) shall be defined:

Table 2-6 Trace, Trace Log, and Trace Event Filter Options: System Trace Events

Event Name	Constant	Associated Data
		Data Type
posix_trace_error	POSIX_TRACE_ERROR	error int
posix_trace_start	POSIX_TRACE_START	event_filter trace_event_set_t
posix_trace_stop	POSIX_TRACE_STOP	auto int
posix_trace_filter	POSIX_TRACE_FILTER	old_event_filter new_event_filter trace_event_set_t
posix_trace_overflow	POSIX_TRACE_OVERFLOW	None.
posix_trace_resume	POSIX_TRACE_RESUME	None.
posix_trace_flush_start	POSIX_TRACE_FLUSH_START	None.
posix_trace_flush_stop	POSIX_TRACE_FLUSH_STOP	None.

2.11.2.2 User Trace Event Type Definitions

The user trace event `POSIX_TRACE_UNNAMED_USEREVENT` is defined in the `<trace.h>` header. If the limit of per-process user trace event names represented by `{TRACE_USER_EVENT_MAX}` has already been reached, this predefined user event shall be returned when the application tries to register more events than allowed. The data associated with this trace event is application-defined.

The following predefined user trace event in Table 2-7 shall be defined:

Table 2-7 Trace Option: User Trace Event

Event Name	Constant
posix_trace_unnamed_userevent	POSIX_TRACE_UNNAMED_USEREVENT

2.11.3 Trace Functions

The trace interface is built and structured to improve portability through use of trace data of opaque type. The object-oriented approach for the manipulation of trace attributes and trace event type identifiers requires definition of many constructor and selector functions which operate on these opaque types. Also, the trace interface must support several different tracing roles. To facilitate reading the trace interface, the trace functions are grouped into small functional sets supporting the three different roles:

- A trace controller process requires functions to set up and customize all the resources needed to run a trace stream, including:
 - Attribute initialization and destruction (`posix_trace_attr_init()`)
 - Identification information manipulation (`posix_trace_attr_getgenversion()`)
 - Trace system behavior modification (`posix_trace_attr_getinherited()`)
 - Trace stream and trace log size set (`posix_trace_attr_getmaxusereventsize()`)

- 3263 — Trace stream creation, flush, and shutdown (*posix_trace_create()*)
- 3264 — Trace stream and trace log clear (*posix_trace_clear()*)
- 3265 — Trace event type identifier manipulation (*posix_trace_trid_eventid_open()*)
- 3266 — Trace event type identifier list exploration (*posix_trace_eventtypelist_getnext_id()*)
- 3267 — Trace event type set manipulation (*posix_trace_eventset_empty()*)
- 3268 — Trace event type filter set (*posix_trace_set_filter()*)
- 3269 — Trace stream start and stop (*posix_trace_start()*)
- 3270 — Trace stream information and status read (*posix_trace_get_attr()*)
- 3271 • A traced process requires functions to instrument trace points:
- 3272 — Trace event type identifiers definition and trace points insertion (*posix_trace_event()*)
- 3273 • A trace analyzer process requires functions to retrieve information from a trace stream and
- 3274 trace log:
- 3275 — Identification information read (*posix_trace_attr_getgenversion()*)
- 3276 — Trace system behavior information read (*posix_trace_attr_getinherited()*)
- 3277 — Trace stream and trace log size get (*posix_trace_attr_getmaxusereventsized()*)
- 3278 — Trace event type identifier manipulation (*posix_trace_trid_eventid_open()*)
- 3279 — Trace event type identifier list exploration (*posix_trace_eventtypelist_getnext_id()*)
- 3280 — Trace log open, rewind, and close (*posix_trace_open()*)
- 3281 — Trace stream information and status read (*posix_trace_get_attr()*)
- 3282 — Trace event read (*posix_trace_getnext_event()*)

3283 2.12 Data Types

3284 All of the data types used by various functions are defined by the implementation. The
 3285 following table describes some of these types. Other types referenced in the description of a
 3286 function, not mentioned here, can be found in the appropriate header for that function.

3287	Defined Type	Description
3288	cc_t	Type used for terminal special characters.
3289	clock_t	Integer or real-floating type used for processor times, as defined in
3290		the ISO C standard.
3291	clockid_t	Used for clock ID type in some timer functions.
3292	dev_t	Arithmetic type used for device numbers.
3293	DIR	Type representing a directory stream.
3294	div_t	Structure type returned by the <i>div()</i> function.
3295	FILE	Structure containing information about a file.
3296	glob_t	Structure type used in pathname pattern matching.
3297	fpos_t	Type containing all information needed to specify uniquely every
3298		

3299
3300
3301
3302
3303
3304
3305
3306
3307
3308
3309
3310
3311
3312
3313
3314
3315
3316
3317
3318
3319
3320
3321
3322
3323
3324
3325
3326
3327
3328
3329
3330
3331
3332
3333
3334
3335
3336
3337
3338
3339
3340
3341
3342
3343
3344
3345
3346
3347

Defined Type	Description
	position within a file.
gid_t	Integer type used for group IDs.
iconv_t	Type used for conversion descriptors.
id_t	Integer type used as a general identifier; can be used to contain at least the largest of a pid_t , uid_t , or gid_t .
ino_t	Unsigned integer type used for file serial numbers.
key_t	Arithmetic type used for XSI interprocess communication.
ldiv_t	Structure type returned by the <i>ldiv()</i> function.
mode_t	Integer type used for file attributes.
mqd_t	Used for message queue descriptors.
nfds_t	Integer type used for the number of file descriptors.
nlink_t	Integer type used for link counts.
off_t	Signed integer type used for file sizes.
pid_t	Signed integer type used for process and process group IDs.
pthread_attr_t	Used to identify a thread attribute object.
pthread_cond_t	Used for condition variables.
pthread_condattr_t	Used to identify a condition attribute object.
pthread_key_t	Used for thread-specific data keys.
pthread_mutex_t	Used for mutexes.
pthread_mutexattr_t	Used to identify a mutex attribute object.
pthread_once_t	Used for dynamic package initialization.
pthread_rwlock_t	Used for read-write locks.
pthread_rwlockattr_t	Used for read-write lock attributes.
pthread_t	Used to identify a thread.
ptrdiff_t	Signed integer type of the result of subtracting two pointers.
regex_t	Structure type used in regular expression matching.
regmatch_t	Structure type used in regular expression matching.
rlim_t	Unsigned integer type used for limit values, to which objects of type int and off_t can be cast without loss of value.
sem_t	Type used in performing semaphore operations.
sig_atomic_t	Integer type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts.
sigset_t	Integer or structure type of an object used to represent sets of signals.
size_t	Unsigned integer type used for size of objects.
speed_t	Type used for terminal baud rates.
ssize_t	Signed integer type used for a count of bytes or an error indication.
suseconds_t	Signed integer type used for time in microseconds.
tcflag_t	Type used for terminal modes.
time_t	Integer or real-floating type used for time in seconds, as defined in the ISO C standard.
timer_t	Used for timer ID returned by the <i>timer_create()</i> function.
uid_t	Integer type used for user IDs.
useconds_t	Unsigned integer type used for time in microseconds.
va_list	Type used for traversing variable argument lists.
wchar_t	Integer type whose range of values can represent distinct codes for

3348
3349
3350
3351
3352
3353
3354
3355

Defined Type	Description
	all members of the largest extended character set specified by the supported locales.
wctype_t	Scalar type which represents a character class descriptor.
wint_t	Integer type capable of storing any valid value of wchar_t or WEOF.
wordexp_t	Structure type used in word expansion.

System Interfaces

3356

3357

3358

This chapter describes the functions, macros, and external variables to support applications portability at the C-language source level.

3359 **NAME**

3360 FD_CLR — macros for synchronous I/O multiplexing

3361 **SYNOPSIS**

3362 #include <sys/time.h>

3363 FD_CLR(int *fd*, fd_set **fdset*);3364 FD_ISSET(int *fd*, fd_set **fdset*);3365 FD_SET(int *fd*, fd_set **fdset*);3366 FD_ZERO(fd_set **fdset*);3367 **DESCRIPTION**3368 Refer to *pselect()*.

3369 **NAME**

3370 _Exit, _exit — terminate a process

3371 **SYNOPSIS**

3372 #include <unistd.h>

3373 void _Exit(int *status*);3374 void _exit(int *status*);3375 **DESCRIPTION**3376 Refer to *exit()*.

3377 NAME

3378 _longjmp, _setjmp — non-local goto

3379 SYNOPSIS

```
3380 xSI      #include <setjmp.h>

3381          void _longjmp(jmp_buf env, int val);
3382          int _setjmp(jmp_buf env);
3383
```

3384 DESCRIPTION

3385 The *_longjmp()* and *_setjmp()* functions shall be equivalent to *longjmp()* and *setjmp()*,
3386 respectively, with the additional restriction that *_longjmp()* and *_setjmp()* shall not manipulate
3387 the signal mask.

3388 If *_longjmp()* is called even though *env* was never initialized by a call to *_setjmp()*, or when the
3389 last such call was in a function that has since returned, the results are undefined.

3390 RETURN VALUE

3391 Refer to *longjmp()* and *setjmp()*.

3392 ERRORS

3393 No errors are defined.

3394 EXAMPLES

3395 None.

3396 APPLICATION USAGE

3397 If *_longjmp()* is executed and the environment in which *_setjmp()* was executed no longer exists,
3398 errors can occur. The conditions under which the environment of the *_setjmp()* no longer exists
3399 include exiting the function that contains the *_setjmp()* call, and exiting an inner block with
3400 temporary storage. This condition might not be detectable, in which case the *_longjmp()* occurs
3401 and, if the environment no longer exists, the contents of the temporary storage of an inner block
3402 are unpredictable. This condition might also cause unexpected process termination. If the
3403 function has returned, the results are undefined.

3404 Passing *longjmp()* a pointer to a buffer not created by *setjmp()*, passing *_longjmp()* a pointer to a
3405 buffer not created by *_setjmp()*, passing *siglongjmp()* a pointer to a buffer not created by
3406 *sigsetjmp()*, or passing any of these three functions a buffer that has been modified by the user
3407 can cause all the problems listed above, and more.

3408 The *_longjmp()* and *_setjmp()* functions are included to support programs written to historical
3409 system interfaces. New applications should use *siglongjmp()* and *sigsetjmp()* respectively.

3410 RATIONALE

3411 None.

3412 FUTURE DIRECTIONS

3413 The *_longjmp()* and *_setjmp()* functions may be marked LEGACY in a future version.

3414 SEE ALSO

3415 *longjmp()*, *setjmp()*, *siglongjmp()*, *sigsetjmp()*, the Base Definitions volume of
3416 IEEE Std 1003.1-2001, <setjmp.h>

3417 CHANGE HISTORY

3418 First released in Issue 4, Version 2.

3419 **Issue 5**

3420 Moved from X/OPEN UNIX extension to BASE.

3421 NAME

3422 _toupper — transliterate uppercase characters to lowercase

3423 SYNOPSIS

3424 xSI #include <ctype.h>

3425 int _tolower(int c);

3426

3427 DESCRIPTION

3428 The *_tolower()* macro shall be equivalent to *tolower(c)* except that the application shall ensure
3429 that the argument *c* is an uppercase letter.

3430 RETURN VALUE

3431 Upon successful completion, *_tolower()* shall return the lowercase letter corresponding to the
3432 argument passed.

3433 ERRORS

3434 No errors are defined.

3435 EXAMPLES

3436 None.

3437 APPLICATION USAGE

3438 None.

3439 RATIONALE

3440 None.

3441 FUTURE DIRECTIONS

3442 None.

3443 SEE ALSO

3444 *tolower()*, *isupper()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale,
3445 <ctype.h>

3446 CHANGE HISTORY

3447 First released in Issue 1. Derived from Issue 1 of the SVID.

3448 Issue 6

3449 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

3450 NAME

3451 _toupper — transliterate lowercase characters to uppercase

3452 SYNOPSIS

3453 xSI #include <ctype.h>

3454 int _toupper(int c);

3455

3456 DESCRIPTION

3457 The *_toupper()* macro shall be equivalent to *toupper()* except that the application shall ensure
3458 that the argument *c* is a lowercase letter.

3459 RETURN VALUE

3460 Upon successful completion, *_toupper()* shall return the uppercase letter corresponding to the
3461 argument passed.

3462 ERRORS

3463 No errors are defined.

3464 EXAMPLES

3465 None.

3466 APPLICATION USAGE

3467 None.

3468 RATIONALE

3469 None.

3470 FUTURE DIRECTIONS

3471 None.

3472 SEE ALSO

3473 *islower()*, *toupper()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale,
3474 <ctype.h>

3475 CHANGE HISTORY

3476 First released in Issue 1. Derived from Issue 1 of the SVID.

3477 Issue 6

3478 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

3479 **NAME**

3480 a64l, l64a — convert between a 32-bit integer and a radix-64 ASCII string

3481 **SYNOPSIS**

3482 xSI #include <stdlib.h>

3483 long a64l(const char *s);

3484 char *l64a(long value);

3485

3486 **DESCRIPTION**

3487 These functions maintain numbers stored in radix-64 ASCII characters. This is a notation by
 3488 which 32-bit integers can be represented by up to six characters; each character represents a digit
 3489 in radix-64 notation. If the type **long** contains more than 32 bits, only the low-order 32 bits shall
 3490 be used for these operations.

3491 The characters used to represent digits are ' .' (dot) for 0, ' / ' for 1, ' 0 ' through ' 9 ' for [2,11],
 3492 ' A ' through ' Z ' for [12,37], and ' a ' through ' z ' for [38,63].

3493 The *a64l()* function shall take a pointer to a radix-64 representation, in which the first digit is the
 3494 least significant, and return the corresponding **long** value. If the string pointed to by *s* contains
 3495 more than six characters, *a64l()* shall use the first six. If the first six characters of the string
 3496 contain a null terminator, *a64l()* shall use only characters preceding the null terminator. The
 3497 *a64l()* function shall scan the character string from left to right with the least significant digit on
 3498 the left, decoding each character as a 6-bit radix-64 number. If the type **long** contains more than
 3499 32 bits, the resulting value is sign-extended. The behavior of *a64l()* is unspecified if *s* is a null
 3500 pointer or the string pointed to by *s* was not generated by a previous call to *l64a()*.

3501 The *l64a()* function shall take a **long** argument and return a pointer to the corresponding radix-
 3502 64 representation. The behavior of *l64a()* is unspecified if *value* is negative.

3503 The value returned by *l64a()* may be a pointer into a static buffer. Subsequent calls to *l64a()* may
 3504 overwrite the buffer.

3505 The *l64a()* function need not be reentrant. A function that is not required to be reentrant is not
 3506 required to be thread-safe.

3507 **RETURN VALUE**

3508 Upon successful completion, *a64l()* shall return the **long** value resulting from conversion of the
 3509 input string. If a string pointed to by *s* is an empty string, *a64l()* shall return 0L.

3510 The *l64a()* function shall return a pointer to the radix-64 representation. If *value* is 0L, *l64a()* shall
 3511 return a pointer to an empty string.

3512 **ERRORS**

3513 No errors are defined.

3514 **EXAMPLES**

3515 None.

3516 **APPLICATION USAGE**3517 If the type **long** contains more than 32 bits, the result of *a64l(l64a(x))* is *x* in the low-order 32 bits.3518 **RATIONALE**3519 This is not the same encoding as used by either encoding variant of the *uuencode* utility.

3520 **FUTURE DIRECTIONS**

3521 None.

3522 **SEE ALSO**

3523 *strtoul()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<stdlib.h>**, the Shell and Utilities
3524 volume of IEEE Std 1003.1-2001, *uuencode*

3525 **CHANGE HISTORY**

3526 First released in Issue 4, Version 2.

3527 **Issue 5**

3528 Moved from X/OPEN UNIX extension to BASE.

3529 Normative text previously in the APPLICATION USAGE section is moved to the
3530 DESCRIPTION.

3531 A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

3532 **NAME**

3533 abort — generate an abnormal process abort

3534 **SYNOPSIS**

3535 #include <stdlib.h>

3536 void abort(void);

3537 **DESCRIPTION**

3538 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 3539 conflict between the requirements described here and the ISO C standard is unintentional. This
 3540 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

3541 The *abort()* function shall cause abnormal process termination to occur, unless the signal
 3542 SIGABRT is being caught and the signal handler does not return.

3543 CX The abnormal termination processing shall include at least the effect of *fclose()* on all open
 3544 streams and the default actions defined for SIGABRT.

3545 XSI On XSI-conformant systems, in addition the abnormal termination processing shall include the
 3546 effect of *fclose()* on message catalog descriptors.

3547 The SIGABRT signal shall be sent to the calling process as if by means of *raise()* with the
 3548 argument SIGABRT.

3549 CX The status made available to *wait()* or *waitpid()* by *abort()* shall be that of a process terminated
 3550 by the SIGABRT signal. The *abort()* function shall override blocking or ignoring the SIGABRT
 3551 signal.

3552 **RETURN VALUE**3553 The *abort()* function shall not return.3554 **ERRORS**

3555 No errors are defined.

3556 **EXAMPLES**

3557 None.

3558 **APPLICATION USAGE**

3559 Catching the signal is intended to provide the application writer with a portable means to abort
 3560 processing, free from possible interference from any implementation-defined functions.

3561 **RATIONALE**

3562 None.

3563 **FUTURE DIRECTIONS**

3564 None.

3565 **SEE ALSO**

3566 *exit()*, *kill()*, *raise()*, *signal()*, *wait()*, *waitpid()*, the Base Definitions volume of
 3567 IEEE Std 1003.1-2001, <stdlib.h>

3568 **CHANGE HISTORY**

3569 First released in Issue 1. Derived from Issue 1 of the SVID.

3570 **Issue 6**

3571 Extensions beyond the ISO C standard are marked.

3572 **NAME**

3573 abs — return an integer absolute value

3574 **SYNOPSIS**

3575 #include <stdlib.h>

3576 int abs(int i);

3577 **DESCRIPTION**

3578 cx The functionality described on this reference page is aligned with the ISO C standard. Any
3579 conflict between the requirements described here and the ISO C standard is unintentional. This
3580 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

3581 The *abs()* function shall compute the absolute value of its integer operand, *i*. If the result cannot
3582 be represented, the behavior is undefined.

3583 **RETURN VALUE**3584 The *abs()* function shall return the absolute value of its integer operand.3585 **ERRORS**

3586 No errors are defined.

3587 **EXAMPLES**

3588 None.

3589 **APPLICATION USAGE**

3590 In two's-complement representation, the absolute value of the negative integer with largest
3591 magnitude {INT_MIN} might not be representable.

3592 **RATIONALE**

3593 None.

3594 **FUTURE DIRECTIONS**

3595 None.

3596 **SEE ALSO**3597 *fabs()*, *labs()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>3598 **CHANGE HISTORY**

3599 First released in Issue 1. Derived from Issue 1 of the SVID.

3600 **Issue 6**

3601 Extensions beyond the ISO C standard are marked.

3602 NAME

3603 accept — accept a new connection on a socket

3604 SYNOPSIS

3605 #include <sys/socket.h>

```
3606 int accept(int socket, struct sockaddr *restrict address,
3607            socklen_t *restrict address_len);
```

3608 DESCRIPTION

3609 The *accept()* function shall extract the first connection on the queue of pending connections,
 3610 create a new socket with the same socket type protocol and address family as the specified
 3611 socket, and allocate a new file descriptor for that socket.

3612 The *accept()* function takes the following arguments:

3613 *socket* Specifies a socket that was created with *socket()*, has been bound to an address
 3614 with *bind()*, and has issued a successful call to *listen()*.

3615 *address* Either a null pointer, or a pointer to a **sockaddr** structure where the address of
 3616 the connecting socket shall be returned.

3617 *address_len* Points to a **socklen_t** structure which on input specifies the length of the
 3618 supplied **sockaddr** structure, and on output specifies the length of the stored
 3619 address.

3620 If *address* is not a null pointer, the address of the peer for the accepted connection shall be stored
 3621 in the **sockaddr** structure pointed to by *address*, and the length of this address shall be stored in
 3622 the object pointed to by *address_len*.

3623 If the actual length of the address is greater than the length of the supplied **sockaddr** structure,
 3624 the stored address shall be truncated.

3625 If the protocol permits connections by unbound clients, and the peer is not bound, then the value
 3626 stored in the object pointed to by *address* is unspecified.

3627 If the listen queue is empty of connection requests and O_NONBLOCK is not set on the file
 3628 descriptor for the socket, *accept()* shall block until a connection is present. If the *listen()* queue is
 3629 empty of connection requests and O_NONBLOCK is set on the file descriptor for the socket,
 3630 *accept()* shall fail and set *errno* to [EAGAIN] or [EWOULDBLOCK].

3631 The accepted socket cannot itself accept more connections. The original socket remains open and
 3632 can accept more connections.

3633 RETURN VALUE

3634 Upon successful completion, *accept()* shall return the non-negative file descriptor of the accepted
 3635 socket. Otherwise, -1 shall be returned and *errno* set to indicate the error.

3636 ERRORS

3637 The *accept()* function shall fail if:

3638 [EAGAIN] or [EWOULDBLOCK]

3639 O_NONBLOCK is set for the socket file descriptor and no connections are
 3640 present to be accepted.

3641 [EBADF] The *socket* argument is not a valid file descriptor.

3642 [ECONNABORTED]

3643 A connection has been aborted.

3644	[EINTR]	The <i>accept()</i> function was interrupted by a signal that was caught before a valid connection arrived.
3645		
3646	[EINVAL]	The <i>socket</i> is not accepting connections.
3647	[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
3648	[ENFILE]	The maximum number of file descriptors in the system are already open.
3649	[ENOTSOCK]	The <i>socket</i> argument does not refer to a socket.
3650	[EOPNOTSUPP]	The <i>socket</i> type of the specified socket does not support accepting connections.
3651		
3652		The <i>accept()</i> function may fail if:
3653	[ENOBUFS]	No buffer space is available.
3654	[ENOMEM]	There was insufficient memory available to complete the operation.
3655	XSR [EPROTO]	A protocol error has occurred; for example, the STREAMS protocol stack has not been initialized.
3656		

3657 EXAMPLES

3658 None.

3659 APPLICATION USAGE

3660 When a connection is available, *select()* indicates that the file descriptor for the socket is ready
3661 for reading.

3662 RATIONALE

3663 None.

3664 FUTURE DIRECTIONS

3665 None.

3666 SEE ALSO

3667 *bind()*, *connect()*, *listen()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-2001,
3668 <sys/socket.h>

3669 CHANGE HISTORY

3670 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

3671 The **restrict** keyword is added to the *accept()* prototype for alignment with the
3672 ISO/IEC 9899:1999 standard.

3673 **NAME**

3674 access — determine accessibility of a file

3675 **SYNOPSIS**

3676 #include <unistd.h>

3677 int access(const char *path, int amode);

3678 **DESCRIPTION**

3679 The *access()* function shall check the file named by the pathname pointed to by the *path*
 3680 argument for accessibility according to the bit pattern contained in *amode*, using the real user ID
 3681 in place of the effective user ID and the real group ID in place of the effective group ID.

3682 The value of *amode* is either the bitwise-inclusive OR of the access permissions to be checked
 3683 (R_OK, W_OK, X_OK) or the existence test (F_OK).

3684 If any access permissions are checked, each shall be checked individually, as described in the
 3685 Base Definitions volume of IEEE Std 1003.1-2001, Chapter 3, Definitions. If the process has
 3686 appropriate privileges, an implementation may indicate success for X_OK even if none of the
 3687 execute file permission bits are set.

3688 **RETURN VALUE**

3689 If the requested access is permitted, *access()* succeeds and shall return 0; otherwise, -1 shall be
 3690 returned and *errno* shall be set to indicate the error.

3691 **ERRORS**3692 The *access()* function shall fail if:

3693 [EACCES] Permission bits of the file mode do not permit the requested access, or search
 3694 permission is denied on a component of the path prefix.

3695 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 3696 argument.

3697 [ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX} or a pathname
 3698 component is longer than {NAME_MAX}.
 3699

3700 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

3701 [ENOTDIR] A component of the path prefix is not a directory.

3702 [EROFS] Write access is requested for a file on a read-only file system.

3703 The *access()* function may fail if:

3704 [EINVAL] The value of the *amode* argument is invalid.

3705 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 3706 resolution of the *path* argument.

3707 [ENAMETOOLONG] As a result of encountering a symbolic link in resolution of the *path* argument,
 3708 the length of the substituted pathname string exceeded {PATH_MAX}.
 3709

3710 [ETXTBSY] Write access is requested for a pure procedure (shared text) file that is being
 3711 executed.

3712 **EXAMPLES**3713 **Testing for the Existence of a File**

3714 The following example tests whether a file named **myfile** exists in the **/tmp** directory.

```
3715 #include <unistd.h>
3716 ...
3717 int result;
3718 const char *filename = "/tmp/myfile";
3719 result = access (filename, F_OK);
```

3720 **APPLICATION USAGE**

3721 Additional values of *amode* other than the set defined in the description may be valid; for
 3722 example, if a system has extended access controls.

3723 **RATIONALE**

3724 In early proposals, some inadequacies in the *access()* function led to the creation of an *eaccess()*
 3725 function because:

- 3726 1. Historical implementations of *access()* do not test file access correctly when the process'
 3727 real user ID is superuser. In particular, they always return zero when testing execute
 3728 permissions without regard to whether the file is executable.
- 3729 2. The superuser has complete access to all files on a system. As a consequence, programs
 3730 started by the superuser and switched to the effective user ID with lesser privileges cannot
 3731 use *access()* to test their file access permissions.

3732 However, the historical model of *eaccess()* does not resolve problem (1), so this volume of
 3733 IEEE Std 1003.1-2001 now allows *access()* to behave in the desired way because several
 3734 implementations have corrected the problem. It was also argued that problem (2) is more easily
 3735 solved by using *open()*, *chdir()*, or one of the *exec* functions as appropriate and responding to the
 3736 error, rather than creating a new function that would not be as reliable. Therefore, *eaccess()* is not
 3737 included in this volume of IEEE Std 1003.1-2001.

3738 The sentence concerning appropriate privileges and execute permission bits reflects the two
 3739 possibilities implemented by historical implementations when checking superuser access for
 3740 *X_OK*.

3741 New implementations are discouraged from returning *X_OK* unless at least one execution
 3742 permission bit is set.

3743 **FUTURE DIRECTIONS**

3744 None.

3745 **SEE ALSO**

3746 *chmod()*, *stat()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<unistd.h>**

3747 **CHANGE HISTORY**

3748 First released in Issue 1. Derived from Issue 1 of the SVID.

3749 **Issue 6**

3750 The following new requirements on POSIX implementations derive from alignment with the
 3751 Single UNIX Specification:

- 3752 • The [ELOOP] mandatory error condition is added.
- 3753 • A second [ENAMETOOLONG] is added as an optional error condition.

- 3754 • The [ETXTBSY] optional error condition is added.

3755 The following changes were made to align with the IEEE P1003.1a draft standard:

- 3756 • The [ELOOP] optional error condition is added.

3757 **NAME**

3758 acos, acosf, acosl — arc cosine functions

3759 **SYNOPSIS**

3760 #include <math.h>

3761 double acos(double x);

3762 float acosf(float x);

3763 long double acosl(long double x);

3764 **DESCRIPTION**

3765 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 3766 conflict between the requirements described here and the ISO C standard is unintentional. This
 3767 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

3768 These functions shall compute the principal value of the arc cosine of their argument *x*. The
 3769 value of *x* should be in the range $[-1,1]$.

3770 An application wishing to check for error situations should set *errno* to zero and call
 3771 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 3772 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 3773 zero, an error has occurred.

3774 **RETURN VALUE**

3775 Upon successful completion, these functions shall return the arc cosine of *x*, in the range $[0,\pi]$
 3776 radians.

3777 **MX** For finite values of *x* not in the range $[-1,1]$, a domain error shall occur, and either a NaN (if
 3778 supported), or an implementation-defined value shall be returned.

3779 **MX** If *x* is NaN, a NaN shall be returned.

3780 If *x* is +1, +0 shall be returned.

3781 If *x* is $\pm\text{Inf}$, a domain error shall occur, and either a NaN (if supported), or an implementation-
 3782 defined value shall be returned.

3783 **ERRORS**

3784 These functions shall fail if:

3785 **MX** Domain Error The *x* argument is finite and is not in the range $[-1,1]$, or is $\pm\text{Inf}$.

3786 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 3787 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling
 3788 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 3789 shall be raised.

3790 **EXAMPLES**

3791 None.

3792 **APPLICATION USAGE**

3793 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
 3794 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

3795 **RATIONALE**

3796 None.

3797 **FUTURE DIRECTIONS**

3798 None.

3799 **SEE ALSO**

3800 *cos()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-2001,
3801 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

3802 **CHANGE HISTORY**

3803 First released in Issue 1. Derived from Issue 1 of the SVID.

3804 **Issue 5**

3805 The DESCRIPTION is updated to indicate how an application should check for an error. This
3806 text was previously published in the APPLICATION USAGE section.

3807 **Issue 6**3808 The *acosf()* and *acosl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

3809 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
3810 revised to align with the ISO/IEC 9899:1999 standard.

3811 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
3812 marked.

3813 **NAME**

3814 acosh, acoshf, acoshl, — inverse hyperbolic cosine functions

3815 **SYNOPSIS**

3816 #include <math.h>

3817 double acosh(double x);

3818 float acoshf(float x);

3819 long double acoshl(long double x);

3820 **DESCRIPTION**

3821 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 3822 conflict between the requirements described here and the ISO C standard is unintentional. This
 3823 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

3824 These functions shall compute the inverse hyperbolic cosine of their argument *x*.

3825 An application wishing to check for error situations should set *errno* to zero and call
 3826 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 3827 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 3828 zero, an error has occurred.

3829 **RETURN VALUE**

3830 Upon successful completion, these functions shall return the inverse hyperbolic cosine of their
 3831 argument.

3832 **MX** For finite values of $x < 1$, a domain error shall occur, and either a NaN (if supported), or an
 3833 implementation-defined value shall be returned.

3834 **MX** If *x* is NaN, a NaN shall be returned.3835 If *x* is +1, +0 shall be returned.3836 If *x* is +Inf, +Inf shall be returned.

3837 If *x* is -Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-
 3838 defined value shall be returned.

3839 **ERRORS**

3840 These functions shall fail if:

3841 **MX** Domain Error The *x* argument is finite and less than +1.0, or is -Inf.

3842 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 3843 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling
 3844 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 3845 shall be raised.

3846 **EXAMPLES**

3847 None.

3848 **APPLICATION USAGE**

3849 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
 3850 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

3851 **RATIONALE**

3852 None.

3853 **FUTURE DIRECTIONS**

3854 None.

3855 **SEE ALSO**

3856 *cosh()*, *feclearexcept()*, *fetestexcept()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section
3857 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

3858 **CHANGE HISTORY**

3859 First released in Issue 4, Version 2.

3860 **Issue 5**

3861 Moved from X/OPEN UNIX extension to BASE.

3862 **Issue 6**3863 The *acosh()* function is no longer marked as an extension.

3864 The *acoshf()*, and *acoshl()* functions are added for alignment with the ISO/IEC 9899:1999
3865 standard.

3866 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
3867 revised to align with the ISO/IEC 9899:1999 standard.

3868 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
3869 marked.

3870 **NAME**

3871 acosl — arc cosine functions

3872 **SYNOPSIS**

3873 #include <math.h>

3874 long double acosl(long double x);

3875 **DESCRIPTION**3876 Refer to *acos()*.

3877 **NAME**3878 aio_cancel — cancel an asynchronous I/O request (**REALTIME**)3879 **SYNOPSIS**3880 AIO `#include <aio.h>`3881 `int aio_cancel(int fildes, struct aiocb *aiocbp);`

3882

3883 **DESCRIPTION**

3884 The *aio_cancel()* function shall attempt to cancel one or more asynchronous I/O requests
 3885 currently outstanding against file descriptor *fildes*. The *aiocbp* argument points to the
 3886 asynchronous I/O control block for a particular request to be canceled. If *aiocbp* is NULL, then
 3887 all outstanding cancelable asynchronous I/O requests against *fildes* shall be canceled.

3888 Normal asynchronous notification shall occur for asynchronous I/O operations that are
 3889 successfully canceled. If there are requests that cannot be canceled, then the normal
 3890 asynchronous completion process shall take place for those requests when they are completed.

3891 For requested operations that are successfully canceled, the associated error status shall be set to
 3892 [ECANCELED] and the return status shall be -1. For requested operations that are not
 3893 successfully canceled, the *aiocbp* shall not be modified by *aio_cancel()*.

3894 If *aiocbp* is not NULL, then if *fildes* does not have the same value as the file descriptor with which
 3895 the asynchronous operation was initiated, unspecified results occur.

3896 Which operations are cancelable is implementation-defined.

3897 **RETURN VALUE**

3898 The *aio_cancel()* function shall return the value AIO_CANCELED to the calling process if the
 3899 requested operation(s) were canceled. The value AIO_NOTCANCELED shall be returned if at
 3900 least one of the requested operation(s) cannot be canceled because it is in progress. In this case,
 3901 the state of the other operations, if any, referenced in the call to *aio_cancel()* is not indicated by
 3902 the return value of *aio_cancel()*. The application may determine the state of affairs for these
 3903 operations by using *aio_error()*. The value AIO_ALLDONE is returned if all of the operations
 3904 have already completed. Otherwise, the function shall return -1 and set *errno* to indicate the
 3905 error.

3906 **ERRORS**

3907 The *aio_cancel()* function shall fail if:

3908 [EBADF] The *fildes* argument is not a valid file descriptor.

3909 **EXAMPLES**

3910 None.

3911 **APPLICATION USAGE**

3912 The *aio_cancel()* function is part of the Asynchronous Input and Output option and need not be
 3913 available on all implementations.

3914 **RATIONALE**

3915 None.

3916 **FUTURE DIRECTIONS**

3917 None.

3918 **SEE ALSO**3919 *aio_read()*, *aio_write()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**aio.h**>3920 **CHANGE HISTORY**

3921 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

3922 **Issue 6**3923 The [ENOSYS] error condition has been removed as stubs need not be provided if an
3924 implementation does not support the Asynchronous Input and Output option.

3925 The APPLICATION USAGE section is added.

3926 **NAME**3927 aio_error — retrieve errors status for an asynchronous I/O operation (**REALTIME**)3928 **SYNOPSIS**

3929 AIO #include <aio.h>

3930 int aio_error(const struct aiocb *aiocbp);

3931

3932 **DESCRIPTION**

3933 The *aio_error()* function shall return the error status associated with the **aiocb** structure
 3934 referenced by the *aiocbp* argument. The error status for an asynchronous I/O operation is the
 3935 SIO *errno* value that would be set by the corresponding *read()*, *write()*, *fdatasync()*, or *fsync()*
 3936 operation. If the operation has not yet completed, then the error status shall be equal to
 3937 [EINPROGRESS].

3938 **RETURN VALUE**

3939 If the asynchronous I/O operation has completed successfully, then 0 shall be returned. If the
 3940 asynchronous operation has completed unsuccessfully, then the error status, as described for
 3941 SIO *read()*, *write()*, *fdatasync()*, and *fsync()*, shall be returned. If the asynchronous I/O operation has
 3942 not yet completed, then [EINPROGRESS] shall be returned.

3943 **ERRORS**3944 The *aio_error()* function may fail if:

3945 [EINVAL] The *aiocbp* argument does not refer to an asynchronous operation whose
 3946 return status has not yet been retrieved.

3947 **EXAMPLES**

3948 None.

3949 **APPLICATION USAGE**

3950 The *aio_error()* function is part of the Asynchronous Input and Output option and need not be
 3951 available on all implementations.

3952 **RATIONALE**

3953 None.

3954 **FUTURE DIRECTIONS**

3955 None.

3956 **SEE ALSO**

3957 *aio_cancel()*, *aio_fsync()*, *aio_read()*, *aio_return()*, *aio_write()*, *close()*, *exec*, *exit()*, *fork()*, *lio_listio()*,
 3958 *lseek()*, *read()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**aio.h**>

3959 **CHANGE HISTORY**

3960 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

3961 **Issue 6**

3962 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 3963 implementation does not support the Asynchronous Input and Output option.

3964 The APPLICATION USAGE section is added.

3965 **NAME**3966 aio_fsync — asynchronous file synchronization (**REALTIME**)3967 **SYNOPSIS**

3968 AIO #include <aio.h>

3969 int aio_fsync(int op, struct aiocb *aiocbp);

3970

3971 **DESCRIPTION**

3972 The *aio_fsync()* function shall asynchronously force all I/O operations associated with the file
 3973 indicated by the file descriptor *aio_fildes* member of the **aiocb** structure referenced by the *aiocbp*
 3974 argument and queued at the time of the call to *aio_fsync()* to the synchronized I/O completion
 3975 state. The function call shall return when the synchronization request has been initiated or
 3976 queued to the file or device (even when the data cannot be synchronized immediately).

3977 If *op* is O_DSYNC, all currently queued I/O operations shall be completed as if by a call to
 3978 *fdatasync()*; that is, as defined for synchronized I/O data integrity completion. If *op* is O_SYNC,
 3979 all currently queued I/O operations shall be completed as if by a call to *fsync()*; that is, as
 3980 defined for synchronized I/O file integrity completion. If the *aio_fsync()* function fails, or if the
 3981 operation queued by *aio_fsync()* fails, then, as for *fsync()* and *fdatasync()*, outstanding I/O
 3982 operations are not guaranteed to have been completed.

3983 If *aio_fsync()* succeeds, then it is only the I/O that was queued at the time of the call to
 3984 *aio_fsync()* that is guaranteed to be forced to the relevant completion state. The completion of
 3985 subsequent I/O on the file descriptor is not guaranteed to be completed in a synchronized
 3986 fashion.

3987 The *aiocbp* argument refers to an asynchronous I/O control block. The *aiocbp* value may be used
 3988 as an argument to *aio_error()* and *aio_return()* in order to determine the error status and return
 3989 status, respectively, of the asynchronous operation while it is proceeding. When the request is
 3990 queued, the error status for the operation is [EINPROGRESS]. When all data has been
 3991 successfully transferred, the error status shall be reset to reflect the success or failure of the
 3992 operation. If the operation does not complete successfully, the error status for the operation shall
 3993 be set to indicate the error. The *aio_sigevent* member determines the asynchronous notification to
 3994 occur as specified in Section 2.4.1 (on page 28) when all operations have achieved synchronized
 3995 I/O completion. All other members of the structure referenced by *aiocbp* are ignored. If the
 3996 control block referenced by *aiocbp* becomes an illegal address prior to asynchronous I/O
 3997 completion, then the behavior is undefined.

3998 If the *aio_fsync()* function fails or *aiocbp* indicates an error condition, data is not guaranteed to
 3999 have been successfully transferred.

4000 **RETURN VALUE**

4001 The *aio_fsync()* function shall return the value 0 to the calling process if the I/O operation is
 4002 successfully queued; otherwise, the function shall return the value -1 and set *errno* to indicate
 4003 the error.

4004 **ERRORS**4005 The *aio_fsync()* function shall fail if:

4006 [EAGAIN] The requested asynchronous operation was not queued due to temporary
 4007 resource limitations.

4008 [EBADF] The *aio_fildes* member of the **aiocb** structure referenced by the *aiocbp* argument
 4009 is not a valid file descriptor open for writing.

4010 [EINVAL] This implementation does not support synchronized I/O for this file.

4011 [EINVAL] A value of *op* other than O_DSYNC or O_SYNC was specified.

4012 In the event that any of the queued I/O operations fail, *aio_fsync()* shall return the error

4013 condition defined for *read()* and *write()*. The error is returned in the error status for the

4014 asynchronous *fsync()* operation, which can be retrieved using *aio_error()*.

4015 **EXAMPLES**

4016 None.

4017 **APPLICATION USAGE**

4018 The *aio_fsync()* function is part of the Asynchronous Input and Output option and need not be

4019 available on all implementations.

4020 **RATIONALE**

4021 None.

4022 **FUTURE DIRECTIONS**

4023 None.

4024 **SEE ALSO**

4025 *fcntl()*, *fdatasync()*, *fsync()*, *open()*, *read()*, *write()*, the Base Definitions volume of

4026 IEEE Std 1003.1-2001, <**aio.h**>

4027 **CHANGE HISTORY**

4028 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

4029 **Issue 6**

4030 The [ENOSYS] error condition has been removed as stubs need not be provided if an

4031 implementation does not support the Asynchronous Input and Output option.

4032 The APPLICATION USAGE section is added.

4033 **NAME**4034 aio_read — asynchronous read from a file (**REALTIME**)4035 **SYNOPSIS**

4036 AIO #include <aio.h>

4037 int aio_read(struct aiocb *aiocbp);

4038

4039 **DESCRIPTION**

4040 The *aio_read()* function shall read *aiocbp->aio_nbytes* from the file associated with
 4041 *aiocbp->aio_fildes* into the buffer pointed to by *aiocbp->aio_buf*. The function call shall return when
 4042 the read request has been initiated or queued to the file or device (even when the data cannot be
 4043 delivered immediately).

4044 PIO If prioritized I/O is supported for this file, then the asynchronous operation shall be submitted
 4045 at a priority equal to the scheduling priority of the process minus *aiocbp->aio_reqprio*.

4046 The *aiocbp* value may be used as an argument to *aio_error()* and *aio_return()* in order to
 4047 determine the error status and return status, respectively, of the asynchronous operation while it
 4048 is proceeding. If an error condition is encountered during queuing, the function call shall return
 4049 without having initiated or queued the request. The requested operation takes place at the
 4050 absolute position in the file as given by *aio_offset*, as if *lseek()* were called immediately prior to
 4051 the operation with an *offset* equal to *aio_offset* and a *whence* equal to *SEEK_SET*. After a
 4052 successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file
 4053 is unspecified.

4054 The *aiocbp->aio_lio_opcode* field shall be ignored by *aio_read()*.

4055 The *aiocbp* argument points to an **aiocb** structure. If the buffer pointed to by *aiocbp->aio_buf* or
 4056 the control block pointed to by *aiocbp* becomes an illegal address prior to asynchronous I/O
 4057 completion, then the behavior is undefined.

4058 Simultaneous asynchronous operations using the same *aiocbp* produce undefined results.

4059 SIO If synchronized I/O is enabled on the file associated with *aiocbp->aio_fildes*, the behavior of this
 4060 function shall be according to the definitions of synchronized I/O data integrity completion and
 4061 synchronized I/O file integrity completion.

4062 For any system action that changes the process memory space while an asynchronous I/O is
 4063 outstanding to the address range being changed, the result of that action is undefined.

4064 For regular files, no data transfer shall occur past the offset maximum established in the open
 4065 file description associated with *aiocbp->aio_fildes*.

4066 **RETURN VALUE**

4067 The *aio_read()* function shall return the value zero to the calling process if the I/O operation is
 4068 successfully queued; otherwise, the function shall return the value -1 and set *errno* to indicate
 4069 the error.

4070 **ERRORS**

4071 The *aio_read()* function shall fail if:

4072 [EAGAIN] The requested asynchronous I/O operation was not queued due to system
 4073 resource limitations.

4074 Each of the following conditions may be detected synchronously at the time of the call to
 4075 *aio_read()*, or asynchronously. If any of the conditions below are detected synchronously, the
 4076 *aio_read()* function shall return -1 and set *errno* to the corresponding value. If any of the
 4077 conditions below are detected asynchronously, the return status of the asynchronous operation

4078 is set to -1, and the error status of the asynchronous operation is set to the corresponding value.

4079 [EBADF] The *aiocbp->aio_fildes* argument is not a valid file descriptor open for reading.

4080 [EINVAL] The file offset value implied by *aiocbp->aio_offset* would be invalid,
 4081 *aiocbp->aio_reqprio* is not a valid value, or *aiocbp->aio_nbytes* is an invalid
 4082 value.

4083 In the case that the *aio_read()* successfully queues the I/O operation but the operation is
 4084 subsequently canceled or encounters an error, the return status of the asynchronous operation is
 4085 one of the values normally returned by the *read()* function call. In addition, the error status of
 4086 the asynchronous operation is set to one of the error statuses normally set by the *read()* function
 4087 call, or one of the following values:

4088 [EBADF] The *aiocbp->aio_fildes* argument is not a valid file descriptor open for reading.

4089 [ECANCELED] The requested I/O was canceled before the I/O completed due to an explicit
 4090 *aio_cancel()* request.

4091 [EINVAL] The file offset value implied by *aiocbp->aio_offset* would be invalid.

4092 The following condition may be detected synchronously or asynchronously:

4093 [EOVERFLOW] The file is a regular file, *aiocbp->aio_nbytes* is greater than 0, and the starting
 4094 offset in *aiocbp->aio_offset* is before the end-of-file and is at or beyond the offset
 4095 maximum in the open file description associated with *aiocbp->aio_fildes*.

4096 EXAMPLES

4097 None.

4098 APPLICATION USAGE

4099 The *aio_read()* function is part of the Asynchronous Input and Output option and need not be
 4100 available on all implementations.

4101 RATIONALE

4102 None.

4103 FUTURE DIRECTIONS

4104 None.

4105 SEE ALSO

4106 *aio_cancel()*, *aio_error()*, *lio_listio()*, *aio_return()*, *aio_write()*, *close()*, *exec*, *exit()*, *fork()*, *lseek()*,
 4107 *read()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**aio.h**>

4108 CHANGE HISTORY

4109 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

4110 Large File Summit extensions are added.

4111 Issue 6

4112 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 4113 implementation does not support the Asynchronous Input and Output option.

4114 The APPLICATION USAGE section is added.

- 4115 The following new requirements on POSIX implementations derive from alignment with the
4116 Single UNIX Specification:
- 4117 • In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file
4118 description. This change is to support large files.
 - 4119 • In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support
4120 large files.

4121 **NAME**4122 aio_return — retrieve return status of an asynchronous I/O operation (**REALTIME**)4123 **SYNOPSIS**

4124 AIO #include <aio.h>

4125 ssize_t aio_return(struct aiocb *aiocbp);

4126

4127 **DESCRIPTION**

4128 The *aio_return()* function shall return the return status associated with the **aiocb** structure
 4129 referenced by the *aiocbp* argument. The return status for an asynchronous I/O operation is the
 4130 value that would be returned by the corresponding *read()*, *write()*, or *fsync()* function call. If the
 4131 error status for the operation is equal to [EINPROGRESS], then the return status for the
 4132 operation is undefined. The *aio_return()* function may be called exactly once to retrieve the
 4133 return status of a given asynchronous operation; thereafter, if the same **aiocb** structure is used in
 4134 a call to *aio_return()* or *aio_error()*, an error may be returned. When the **aiocb** structure referred
 4135 to by *aiocbp* is used to submit another asynchronous operation, then *aio_return()* may be
 4136 successfully used to retrieve the return status of that operation.

4137 **RETURN VALUE**

4138 If the asynchronous I/O operation has completed, then the return status, as described for *read()*,
 4139 *write()*, and *fsync()*, shall be returned. If the asynchronous I/O operation has not yet completed,
 4140 the results of *aio_return()* are undefined.

4141 **ERRORS**4142 The *aio_return()* function may fail if:

4143 [EINVAL] The *aiocbp* argument does not refer to an asynchronous operation whose
 4144 return status has not yet been retrieved.

4145 **EXAMPLES**

4146 None.

4147 **APPLICATION USAGE**

4148 The *aio_return()* function is part of the Asynchronous Input and Output option and need not be
 4149 available on all implementations.

4150 **RATIONALE**

4151 None.

4152 **FUTURE DIRECTIONS**

4153 None.

4154 **SEE ALSO**

4155 *aio_cancel()*, *aio_error()*, *aio_fsync()*, *aio_read()*, *aio_write()*, *close()*, *exec*, *exit()*, *fork()*, *lio_listio()*,
 4156 *lseek()*, *read()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**aio.h**>

4157 **CHANGE HISTORY**

4158 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

4159 **Issue 6**

4160 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 4161 implementation does not support the Asynchronous Input and Output option.

4162 The APPLICATION USAGE section is added.

4163 The [EINVAL] error condition is updated as a “may fail”. This is for consistency with the
 4164 DESCRIPTION.

4165 **NAME**4166 aio_suspend — wait for an asynchronous I/O request (**REALTIME**)4167 **SYNOPSIS**4168 AIO `#include <aio.h>`

```
4169 int aio_suspend(const struct aiocb * const list[], int nent,
4170               const struct timespec *timeout);
4171
```

4172 **DESCRIPTION**

4173 The *aio_suspend()* function shall suspend the calling thread until at least one of the asynchronous
 4174 I/O operations referenced by the *list* argument has completed, until a signal interrupts the
 4175 function, or, if *timeout* is not NULL, until the time interval specified by *timeout* has passed. If any
 4176 of the **aiocb** structures in the list correspond to completed asynchronous I/O operations (that is,
 4177 the error status for the operation is not equal to [EINPROGRESS]) at the time of the call, the
 4178 function shall return without suspending the calling thread. The *list* argument is an array of
 4179 pointers to asynchronous I/O control blocks. The *nent* argument indicates the number of
 4180 elements in the array. Each **aiocb** structure pointed to has been used in initiating an
 4181 asynchronous I/O request via *aio_read()*, *aio_write()*, or *lio_listio()*. This array may contain
 4182 NULL pointers, which are ignored. If this array contains pointers that refer to **aiocb** structures
 4183 that have not been used in submitting asynchronous I/O, the effect is undefined.

4184 If the time interval indicated in the **timespec** structure pointed to by *timeout* passes before any of
 4185 the I/O operations referenced by *list* are completed, then *aio_suspend()* shall return with an
 4186 error. If the Monotonic Clock option is supported, the clock that shall be used to measure this
 4187 time interval shall be the CLOCK_MONOTONIC clock.

4188 **RETURN VALUE**

4189 If the *aio_suspend()* function returns after one or more asynchronous I/O operations have
 4190 completed, the function shall return zero. Otherwise, the function shall return a value of -1 and
 4191 set *errno* to indicate the error.

4192 The application may determine which asynchronous I/O completed by scanning the associated
 4193 error and return status using *aio_error()* and *aio_return()*, respectively.

4194 **ERRORS**4195 The *aio_suspend()* function shall fail if:

4196 [EAGAIN] No asynchronous I/O indicated in the list referenced by *list* completed in the
 4197 time interval indicated by *timeout*.

4198 [EINTR] A signal interrupted the *aio_suspend()* function. Note that, since each
 4199 asynchronous I/O operation may possibly provoke a signal when it
 4200 completes, this error return may be caused by the completion of one (or more)
 4201 of the very I/O operations being awaited.

4202 **EXAMPLES**

4203 None.

4204 **APPLICATION USAGE**

4205 The *aio_suspend()* function is part of the Asynchronous Input and Output option and need not
 4206 be available on all implementations.

4207 **RATIONALE**

4208 None.

4209 FUTURE DIRECTIONS

4210 None.

4211 SEE ALSO

4212 *aio_read()*, *aio_write()*, *lio_listio()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**aio.h**>

4213 CHANGE HISTORY

4214 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

4215 Issue 6

4216 The [ENOSYS] error condition has been removed as stubs need not be provided if an
4217 implementation does not support the Asynchronous Input and Output option.

4218 The APPLICATION USAGE section is added.

4219 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that the
4220 CLOCK_MONOTONIC clock, if supported, is used.

4221 **NAME**4222 aio_write — asynchronous write to a file (**REALTIME**)4223 **SYNOPSIS**4224 AIO `#include <aio.h>`4225 `int aio_write(struct aiocb *aiocbp);`

4226

4227 **DESCRIPTION**

4228 The *aio_write()* function shall write *aiocbp->aio_nbytes* to the file associated with *aiocbp->aio_fildes*
 4229 from the buffer pointed to by *aiocbp->aio_buf*. The function shall return when the write request
 4230 has been initiated or, at a minimum, queued to the file or device.

4231 PIO If prioritized I/O is supported for this file, then the asynchronous operation shall be submitted
 4232 at a priority equal to the scheduling priority of the process minus *aiocbp->aio_reqprio*.

4233 The *aiocbp* argument may be used as an argument to *aio_error()* and *aio_return()* in order to
 4234 determine the error status and return status, respectively, of the asynchronous operation while it
 4235 is proceeding.

4236 The *aiocbp* argument points to an **aiocb** structure. If the buffer pointed to by *aiocbp->aio_buf* or
 4237 the control block pointed to by *aiocbp* becomes an illegal address prior to asynchronous I/O
 4238 completion, then the behavior is undefined.

4239 If O_APPEND is not set for the file descriptor *aio_fildes*, then the requested operation shall take
 4240 place at the absolute position in the file as given by *aio_offset*, as if *lseek()* were called
 4241 immediately prior to the operation with an *offset* equal to *aio_offset* and a *whence* equal to
 4242 SEEK_SET. If O_APPEND is set for the file descriptor, write operations append to the file in the
 4243 same order as the calls were made. After a successful call to enqueue an asynchronous I/O
 4244 operation, the value of the file offset for the file is unspecified.

4245 The *aiocbp->aio_lio_opcode* field shall be ignored by *aio_write()*.

4246 Simultaneous asynchronous operations using the same *aiocbp* produce undefined results.

4247 SIO If synchronized I/O is enabled on the file associated with *aiocbp->aio_fildes*, the behavior of this
 4248 function shall be according to the definitions of synchronized I/O data integrity completion, and
 4249 synchronized I/O file integrity completion.

4250 For any system action that changes the process memory space while an asynchronous I/O is
 4251 outstanding to the address range being changed, the result of that action is undefined.

4252 For regular files, no data transfer shall occur past the offset maximum established in the open
 4253 file description associated with *aiocbp->aio_fildes*.

4254 **RETURN VALUE**

4255 The *aio_write()* function shall return the value zero to the calling process if the I/O operation is
 4256 successfully queued; otherwise, the function shall return the value -1 and set *errno* to indicate
 4257 the error.

4258 **ERRORS**

4259 The *aio_write()* function shall fail if:

4260 [EAGAIN] The requested asynchronous I/O operation was not queued due to system
 4261 resource limitations.

4262 Each of the following conditions may be detected synchronously at the time of the call to
 4263 *aio_write()*, or asynchronously. If any of the conditions below are detected synchronously, the
 4264 *aio_write()* function shall return -1 and set *errno* to the corresponding value. If any of the

conditions below are detected asynchronously, the return status of the asynchronous operation shall be set to -1, and the error status of the asynchronous operation is set to the corresponding value.

[EBADF] The *aiochp->aio_fildes* argument is not a valid file descriptor open for writing.

[EINVAL] The file offset value implied by *aiochp->aio_offset* would be invalid, *aiochp->aio_reqprio* is not a valid value, or *aiochp->aio_nbytes* is an invalid value.

In the case that the *aio_write()* successfully queues the I/O operation, the return status of the asynchronous operation shall be one of the values normally returned by the *write()* function call. If the operation is successfully queued but is subsequently canceled or encounters an error, the error status for the asynchronous operation contains one of the values normally set by the *write()* function call, or one of the following:

[EBADF] The *aiochp->aio_fildes* argument is not a valid file descriptor open for writing.

[EINVAL] The file offset value implied by *aiochp->aio_offset* would be invalid.

[ECANCELED] The requested I/O was canceled before the I/O completed due to an explicit *aio_cancel()* request.

The following condition may be detected synchronously or asynchronously:

[EFBIG] The file is a regular file, *aiochp->aio_nbytes* is greater than 0, and the starting offset in *aiochp->aio_offset* is at or beyond the offset maximum in the open file description associated with *aiochp->aio_fildes*.

EXAMPLES

None.

APPLICATION USAGE

The *aio_write()* function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

aio_cancel(), *aio_error()*, *aio_read()*, *aio_return()*, *close()*, *exec*, *exit()*, *fork()*, *lio_listio()*, *lseek()*, *write()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**aio.h**>

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Large File Summit extensions are added.

Issue 6

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Asynchronous Input and Output option.

The APPLICATION USAGE section is added.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, text is added to indicate that for regular files no data transfer occurs past the offset maximum established in the open file description associated with

4308 *aiocbp->aio_fildes*.

- 4309 • The [EFBIG] error is added as part of the large file support extensions.

4310 NAME

4311 alarm — schedule an alarm signal

4312 SYNOPSIS

4313 #include <unistd.h>

4314 unsigned alarm(unsigned *seconds*);

4315 DESCRIPTION

4316 The *alarm()* function shall cause the system to generate a SIGALRM signal for the process after
 4317 the number of realtime seconds specified by *seconds* have elapsed. Processor scheduling delays
 4318 may prevent the process from handling the signal as soon as it is generated.

4319 If *seconds* is 0, a pending alarm request, if any, is canceled.

4320 Alarm requests are not stacked; only one SIGALRM generation can be scheduled in this manner.
 4321 If the SIGALRM signal has not yet been generated, the call shall result in rescheduling the time
 4322 at which the SIGALRM signal is generated.

4323 XSI Interactions between *alarm()* and any of *setitimer()*, *ualarm()*, or *usleep()* are unspecified.

4324 RETURN VALUE

4325 If there is a previous *alarm()* request with time remaining, *alarm()* shall return a non-zero value
 4326 that is the number of seconds until the previous request would have generated a SIGALRM
 4327 signal. Otherwise, *alarm()* shall return 0.

4328 ERRORS

4329 The *alarm()* function is always successful, and no return value is reserved to indicate an error.

4330 EXAMPLES

4331 None.

4332 APPLICATION USAGE

4333 The *fork()* function clears pending alarms in the child process. A new process image created by
 4334 one of the *exec* functions inherits the time left to an alarm signal in the old process' image.

4335 Application writers should note that the type of the argument *seconds* and the return value of
 4336 *alarm()* is **unsigned**. That means that a Strictly Conforming POSIX System Interfaces
 4337 Application cannot pass a value greater than the minimum guaranteed value for {UINT_MAX},
 4338 which the ISO C standard sets as 65 535, and any application passing a larger value is restricting
 4339 its portability. A different type was considered, but historical implementations, including those
 4340 with a 16-bit **int** type, consistently use either **unsigned** or **int**.

4341 Application writers should be aware of possible interactions when the same process uses both
 4342 the *alarm()* and *sleep()* functions.

4343 RATIONALE

4344 Many historical implementations (including Version 7 and System V) allow an alarm to occur up
 4345 to a second early. Other implementations allow alarms up to half a second or one clock tick
 4346 early or do not allow them to occur early at all. The latter is considered most appropriate, since it
 4347 gives the most predictable behavior, especially since the signal can always be delayed for an
 4348 indefinite amount of time due to scheduling. Applications can thus choose the *seconds* argument
 4349 as the minimum amount of time they wish to have elapse before the signal.

4350 The term “realtime” here and elsewhere (*sleep()*, *times()*) is intended to mean “wall clock” time
 4351 as common English usage, and has nothing to do with “realtime operating systems”. It is in
 4352 contrast to *virtual time*, which could be misinterpreted if just *time* were used.

4353 In some implementations, including 4.3 BSD, very large values of the *seconds* argument are
 4354 silently rounded down to an implementation-defined maximum value. This maximum is large

4355 enough (to the order of several months) that the effect is not noticeable.

4356 There were two possible choices for alarm generation in multi-threaded applications: generation
4357 for the calling thread or generation for the process. The first option would not have been
4358 particularly useful since the alarm state is maintained on a per-process basis and the alarm that
4359 is established by the last invocation of *alarm()* is the only one that would be active.

4360 Furthermore, allowing generation of an asynchronous signal for a thread would have introduced
4361 an exception to the overall signal model. This requires a compelling reason in order to be
4362 justified.

4363 **FUTURE DIRECTIONS**

4364 None.

4365 **SEE ALSO**

4366 *alarm()*, *exec*, *fork()*, *getitimer()*, *pause()*, *sigaction()*, *sleep()*, *ualarm()*, *usleep()*, the Base
4367 Definitions volume of IEEE Std 1003.1-2001, <signal.h>, <unistd.h>

4368 **CHANGE HISTORY**

4369 First released in Issue 1. Derived from Issue 1 of the SVID.

4370 **Issue 6**

4371 The following new requirements on POSIX implementations derive from alignment with the
4372 Single UNIX Specification:

- 4373 • The DESCRIPTION is updated to indicate that interactions with the *setitimer()*, *ualarm()*, and
4374 *usleep()* functions are unspecified.

4375 NAME

4376 asctime, asctime_r — convert date and time to a string

4377 SYNOPSIS

4378 #include <time.h>

4379 char *asctime(const struct tm *timeptr);

4380 TSF char *asctime_r(const struct tm *restrict tm, char *restrict buf);

4381

4382 DESCRIPTION

4383 CX For *asctime()*: The functionality described on this reference page is aligned with the ISO C
 4384 standard. Any conflict between the requirements described here and the ISO C standard is
 4385 unintentional. This volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

4386 The *asctime()* function shall convert the broken-down time in the structure pointed to by *timeptr*
 4387 into a string in the form:

4388 Sun Sep 16 01:03:52 1973\n\0

4389 using the equivalent of the following algorithm:

```

4390 char *asctime(const struct tm *timeptr)
4391 {
4392     static char wday_name[7][3] = {
4393         "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
4394     };
4395     static char mon_name[12][3] = {
4396         "Jan", "Feb", "Mar", "Apr", "May", "Jun",
4397         "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
4398     };
4399     static char result[26];

4400     sprintf(result, "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",
4401         wday_name[timeptr->tm_wday],
4402         mon_name[timeptr->tm_mon],
4403         timeptr->tm_mday, timeptr->tm_hour,
4404         timeptr->tm_min, timeptr->tm_sec,
4405         1900 + timeptr->tm_year);
4406     return result;
4407 }
```

4408 The **tm** structure is defined in the <time.h> header.

4409 CX The *asctime()*, *ctime()*, *gmtime()*, and *localtime()* functions shall return values in one of two static
 4410 objects: a broken-down time structure and an array of type **char**. Execution of any of the
 4411 functions may overwrite the information returned in either of these objects by any of the other
 4412 functions.

4413 The *asctime()* function need not be reentrant. A function that is not required to be reentrant is not
 4414 required to be thread-safe.

4415 TSF The *asctime_r()* function shall convert the broken-down time in the structure pointed to by *tm*
 4416 into a string (of the same form as that returned by *asctime()*) that is placed in the user-supplied
 4417 buffer pointed to by *buf* (which shall contain at least 26 bytes) and then return *buf*.

4418 RETURN VALUE

4419 Upon successful completion, *asctime()* shall return a pointer to the string.

4420 TSF Upon successful completion, *asctime_r()* shall return a pointer to a character string containing
4421 the date and time. This string is pointed to by the argument *buf*. If the function is unsuccessful,
4422 it shall return NULL.

4423 ERRORS

4424 No errors are defined.

4425 EXAMPLES

4426 None.

4427 APPLICATION USAGE

4428 Values for the broken-down time structure can be obtained by calling *gmtime()* or *localtime()*.
4429 This function is included for compatibility with older implementations, and does not support
4430 localized date and time formats. Applications should use *strftime()* to achieve maximum
4431 portability.

4432 The *asctime_r()* function is thread-safe and shall return values in a user-supplied buffer instead
4433 of possibly using a static data area that may be overwritten by each call.

4434 RATIONALE

4435 None.

4436 FUTURE DIRECTIONS

4437 None.

4438 SEE ALSO

4439 *clock()*, *ctime()*, *difftime()*, *gmtime()*, *localtime()*, *mktime()*, *strftime()*, *strptime()*, *time()*, *utime()*,
4440 the Base Definitions volume of IEEE Std 1003.1-2001, <**time.h**>

4441 CHANGE HISTORY

4442 First released in Issue 1. Derived from Issue 1 of the SVID.

4443 Issue 5

4444 Normative text previously in the APPLICATION USAGE section is moved to the
4445 DESCRIPTION.

4446 The *asctime_r()* function is included for alignment with the POSIX Threads Extension.

4447 A note indicating that the *asctime()* function need not be reentrant is added to the
4448 DESCRIPTION.

4449 Issue 6

4450 The *asctime_r()* function is marked as part of the Thread-Safe Functions option.

4451 Extensions beyond the ISO C standard are marked.

4452 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
4453 its avoidance of possibly using a static data area.

4454 The DESCRIPTION of *asctime_r()* is updated to describe the format of the string returned.

4455 The **restrict** keyword is added to the *asctime_r()* prototype for alignment with the
4456 ISO/IEC 9899:1999 standard.

4457 **NAME**4458 `asin, asinf, asinl` — arc sine function4459 **SYNOPSIS**4460 `#include <math.h>`4461 `double asin(double x);`4462 `float asinf(float x);`4463 `long double asinl(long double x);`4464 **DESCRIPTION**

4465 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 4466 conflict between the requirements described here and the ISO C standard is unintentional. This
 4467 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

4468 These functions shall compute the principal value of the arc sine of their argument *x*. The value
 4469 of *x* should be in the range $[-1,1]$.

4470 An application wishing to check for error situations should set *errno* to zero and call
 4471 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 4472 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 4473 zero, an error has occurred.

4474 **RETURN VALUE**

4475 Upon successful completion, these functions shall return the arc sine of *x*, in the range
 4476 $[-\pi/2, \pi/2]$ radians.

4477 **MX** For finite values of *x* not in the range $[-1,1]$, a domain error shall occur, and either a NaN (if
 4478 supported), or an implementation-defined value shall be returned.

4479 **MX** If *x* is NaN, a NaN shall be returned.

4480 If *x* is ± 0 , *x* shall be returned.

4481 If *x* is $\pm \text{Inf}$, a domain error shall occur, and either a NaN (if supported), or an implementation-
 4482 defined value shall be returned.

4483 If *x* is subnormal, a range error may occur and *x* should be returned.

4484 **ERRORS**

4485 These functions shall fail if:

4486 **MX** **Domain Error** The *x* argument is finite and is not in the range $[-1,1]$, or is $\pm \text{Inf}$.

4487 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 4488 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling
 4489 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 4490 shall be raised.

4491 These functions may fail if:

4492 **MX** **Range Error** The value of *x* is subnormal.

4493 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 4494 then *errno* shall be set to [ERANGE]. If the integer expression
 4495 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
 4496 floating-point exception shall be raised.

4497 **EXAMPLES**

4498 None.

4499 **APPLICATION USAGE**

4500 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
4501 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

4502 **RATIONALE**

4503 None.

4504 **FUTURE DIRECTIONS**

4505 None.

4506 **SEE ALSO**

4507 *feclearexcept()*, *fetetestexcept()*, *isnan()*, *sin()*, the Base Definitions volume of IEEE Std 1003.1-2001,
4508 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

4509 **CHANGE HISTORY**

4510 First released in Issue 1. Derived from Issue 1 of the SVID.

4511 **Issue 5**

4512 The DESCRIPTION is updated to indicate how an application should check for an error. This
4513 text was previously published in the APPLICATION USAGE section.

4514 **Issue 6**4515 The *asinf()* and *asinl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

4516 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
4517 revised to align with the ISO/IEC 9899:1999 standard.

4518 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
4519 marked.

4520 **NAME**

4521 asinh, asinhf, asinhl — inverse hyperbolic sine functions

4522 **SYNOPSIS**

4523 #include <math.h>

4524 double asinh(double x);

4525 float asinhf(float x);

4526 long double asinhl(long double x);

4527 **DESCRIPTION**

4528 CX The functionality described on this reference page is aligned with the ISO C standard. Any
4529 conflict between the requirements described here and the ISO C standard is unintentional. This
4530 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

4531 These functions shall compute the inverse hyperbolic sine of their argument *x*.

4532 An application wishing to check for error situations should set *errno* to zero and call
4533 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
4534 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
4535 zero, an error has occurred.

4536 **RETURN VALUE**

4537 Upon successful completion, these functions shall return the inverse hyperbolic sine of their
4538 argument.

4539 MX If *x* is NaN, a NaN shall be returned.

4540 If *x* is ± 0 , or $\pm \text{Inf}$, *x* shall be returned.

4541 If *x* is subnormal, a range error may occur and *x* should be returned.

4542 **ERRORS**

4543 These functions may fail if:

4544 MX **Range Error** The value of *x* is subnormal.

4545 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
4546 then *errno* shall be set to [ERANGE]. If the integer expression
4547 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
4548 floating-point exception shall be raised.

4549 **EXAMPLES**

4550 None.

4551 **APPLICATION USAGE**

4552 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
4553 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

4554 **RATIONALE**

4555 None.

4556 **FUTURE DIRECTIONS**

4557 None.

4558 **SEE ALSO**

4559 *feclearexcept*(), *fetestexcept*(), *sinh*(), the Base Definitions volume of IEEE Std 1003.1-2001, Section
4560 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

4561 **CHANGE HISTORY**

4562 First released in Issue 4, Version 2.

4563 **Issue 5**

4564 Moved from X/OPEN UNIX extension to BASE.

4565 **Issue 6**4566 The *asinh()* function is no longer marked as an extension.4567 The *asinhf()* and *asinhll()* functions are added for alignment with the ISO/IEC 9899:1999
4568 standard.4569 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
4570 revised to align with the ISO/IEC 9899:1999 standard.4571 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
4572 marked.

4573 **NAME**4574 **asinl** — arc sine function4575 **SYNOPSIS**4576 `#include <math.h>`4577 `long double asinl(long double x);`4578 **DESCRIPTION**4579 Refer to *asin()*.

4580 **NAME**

4581 assert — insert program diagnostics

4582 **SYNOPSIS**

4583 #include <assert.h>

4584 void assert(*scalar expression*);4585 **DESCRIPTION**

4586 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
4587 conflict between the requirements described here and the ISO C standard is unintentional. This
4588 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

4589 The *assert()* macro shall insert diagnostics into programs; it shall expand to a **void** expression.
4590 When it is executed, if *expression* (which shall have a **scalar** type) is false (that is, compares equal
4591 to 0), *assert()* shall write information about the particular call that failed on *stderr* and shall call
4592 *abort()*.

4593 The information written about the call that failed shall include the text of the argument, the
4594 name of the source file, the source file line number, and the name of the enclosing function; the
4595 latter are, respectively, the values of the preprocessing macros `__FILE__` and `__LINE__` and of
4596 the identifier `__func__`.

4597 Forcing a definition of the name `NDEBUG`, either from the compiler command line or with the
4598 preprocessor control statement `#define NDEBUG` ahead of the `#include <assert.h>` statement,
4599 shall stop assertions from being compiled into the program.

4600 **RETURN VALUE**4601 The *assert()* macro shall not return a value.4602 **ERRORS**

4603 No errors are defined.

4604 **EXAMPLES**

4605 None.

4606 **APPLICATION USAGE**

4607 None.

4608 **RATIONALE**

4609 None.

4610 **FUTURE DIRECTIONS**

4611 None.

4612 **SEE ALSO**4613 *abort()*, *stderr*, the Base Definitions volume of IEEE Std 1003.1-2001, `<assert.h>`4614 **CHANGE HISTORY**

4615 First released in Issue 1. Derived from Issue 1 of the SVID.

4616 **Issue 6**

4617 The prototype for the *expression* argument to *assert()* is changed from **int** to **scalar** for alignment
4618 with the ISO/IEC 9899:1999 standard.

4619 The DESCRIPTION of *assert()* is updated for alignment with the ISO/IEC 9899:1999 standard.

4620 **NAME**

4621 atan, atanf, atanl — arc tangent function

4622 **SYNOPSIS**

4623 #include <math.h>

4624 double atan(double x);

4625 float atanf(float x);

4626 long double atanl(long double x);

4627 **DESCRIPTION**

4628 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 4629 conflict between the requirements described here and the ISO C standard is unintentional. This
 4630 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

4631 These functions shall compute the principal value of the arc tangent of their argument *x*.

4632 An application wishing to check for error situations should set *errno* to zero and call
 4633 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 4634 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 4635 zero, an error has occurred.

4636 **RETURN VALUE**

4637 Upon successful completion, these functions shall return the arc tangent of *x* in the range
 4638 $[-\pi/2, \pi/2]$ radians.

4639 MX If *x* is NaN, a NaN shall be returned.4640 If *x* is ± 0 , *x* shall be returned.4641 If *x* is $\pm \text{Inf}$, $\pm \pi/2$ shall be returned.4642 If *x* is subnormal, a range error may occur and *x* should be returned.4643 **ERRORS**

4644 These functions may fail if:

4645 MX **Range Error** The value of *x* is subnormal.

4646 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 4647 then *errno* shall be set to [ERANGE]. If the integer expression
 4648 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
 4649 floating-point exception shall be raised.

4650 **EXAMPLES**

4651 None.

4652 **APPLICATION USAGE**

4653 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
 4654 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

4655 **RATIONALE**

4656 None.

4657 **FUTURE DIRECTIONS**

4658 None.

4659 **SEE ALSO**

4660 *atan2()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, *tan()*, the Base Definitions volume of
 4661 IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
 4662 <math.h>

4663 **CHANGE HISTORY**

4664 First released in Issue 1. Derived from Issue 1 of the SVID.

4665 **Issue 5**4666 The DESCRIPTION is updated to indicate how an application should check for an error. This
4667 text was previously published in the APPLICATION USAGE section.4668 **Issue 6**4669 The *atanf()* and *atanl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.4670 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
4671 revised to align with the ISO/IEC 9899:1999 standard.4672 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
4673 marked.

4674 **NAME**

4675 atan2, atan2f, atan2l — arc tangent functions

4676 **SYNOPSIS**

4677 #include <math.h>

4678 double atan2(double y, double x);

4679 float atan2f(float y, float x);

4680 long double atan2l(long double y, long double x);

4681 **DESCRIPTION**

4682 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 4683 conflict between the requirements described here and the ISO C standard is unintentional. This
 4684 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

4685 These functions shall compute the principal value of the arc tangent of y/x , using the signs of
 4686 both arguments to determine the quadrant of the return value.

4687 An application wishing to check for error situations should set *errno* to zero and call
 4688 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 4689 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 4690 zero, an error has occurred.

4691 **RETURN VALUE**

4692 Upon successful completion, these functions shall return the arc tangent of y/x in the range
 4693 $[-\pi, \pi]$ radians.

4694 If y is ± 0 and x is < 0 , $\pm\pi$ shall be returned.4695 If y is ± 0 and x is > 0 , ± 0 shall be returned.4696 If y is < 0 and x is ± 0 , $-\pi/2$ shall be returned.4697 If y is > 0 and x is ± 0 , $\pi/2$ shall be returned.4698 If x is 0, a pole error shall not occur.4699 **MX** If either x or y is NaN, a NaN shall be returned.4700 If the result underflows, a range error may occur and y/x should be returned.4701 If y is ± 0 and x is -0 , $\pm\pi$ shall be returned.4702 If y is ± 0 and x is $+0$, ± 0 shall be returned.4703 For finite values of $\pm y > 0$, if x is $-\text{Inf}$, $\pm\pi$ shall be returned.4704 For finite values of $\pm y > 0$, if x is $+\text{Inf}$, ± 0 shall be returned.4705 For finite values of x , if y is $\pm\text{Inf}$, $\pm\pi/2$ shall be returned.4706 If y is $\pm\text{Inf}$ and x is $-\text{Inf}$, $\pm 3\pi/4$ shall be returned.4707 If y is $\pm\text{Inf}$ and x is $+\text{Inf}$, $\pm\pi/4$ shall be returned.

4708 If both arguments are 0, a domain error shall not occur.

4709 **ERRORS**

4710 These functions may fail if:

4711 **MX** **Range Error** The result underflows.

4712 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 4713 then *errno* shall be set to [ERANGE]. If the integer expression

4714 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
4715 floating-point exception shall be raised.

4716 **EXAMPLES**

4717 None.

4718 **APPLICATION USAGE**

4719 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
4720 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

4721 **RATIONALE**

4722 None.

4723 **FUTURE DIRECTIONS**

4724 None.

4725 **SEE ALSO**

4726 *atan()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, *tan()*, the Base Definitions volume of
4727 IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
4728 <math.h>

4729 **CHANGE HISTORY**

4730 First released in Issue 1. Derived from Issue 1 of the SVID.

4731 **Issue 5**

4732 The DESCRIPTION is updated to indicate how an application should check for an error. This
4733 text was previously published in the APPLICATION USAGE section.

4734 **Issue 6**

4735 The *atan2f()* and *atan2l()* functions are added for alignment with the ISO/IEC 9899:1999
4736 standard.

4737 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
4738 revised to align with the ISO/IEC 9899:1999 standard.

4739 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
4740 marked.

4741 **NAME**

4742 atanf — arc tangent function

4743 **SYNOPSIS**

4744 #include <math.h>

4745 float atanf(float x);

4746 **DESCRIPTION**4747 Refer to *atan()*.

4748 **NAME**

4749 atanh, atanhf, atanh1 — inverse hyperbolic tangent functions

4750 **SYNOPSIS**

4751 #include <math.h>

4752 double atanh(double x);

4753 float atanhf(float x);

4754 long double atanh1(long double x);

4755 **DESCRIPTION**

4756 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 4757 conflict between the requirements described here and the ISO C standard is unintentional. This
 4758 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

4759 These functions shall compute the inverse hyperbolic tangent of their argument *x*.

4760 An application wishing to check for error situations should set *errno* to zero and call
 4761 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 4762 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 4763 zero, an error has occurred.

4764 **RETURN VALUE**4765 Upon successful completion, these functions shall return the inverse hyperbolic tangent of their
4766 argument.

4767 If *x* is ± 1 , a pole error shall occur, and *atanh()*, *atanhf()*, and *atanh1()* shall return the value of the
 4768 macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively, with the same sign as the
 4769 correct value of the function.

4770 **MX** For finite $|x| > 1$, a domain error shall occur, and either a NaN (if supported), or an
 4771 implementation-defined value shall be returned.

4772 **MX** If *x* is NaN, a NaN shall be returned.4773 If *x* is ± 0 , *x* shall be returned.

4774 If *x* is $\pm \text{Inf}$, a domain error shall occur, and either a NaN (if supported), or an implementation-
 4775 defined value shall be returned.

4776 If *x* is subnormal, a range error may occur and *x* should be returned.4777 **ERRORS**

4778 These functions shall fail if:

4779 **MX** Domain Error The *x* argument is finite and not in the range $[-1, 1]$, or is $\pm \text{Inf}$.

4780 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 4781 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling
 4782 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 4783 shall be raised.

4784 Pole Error The *x* argument is ± 1 .

4785 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 4786 then *errno* shall be set to [ERANGE]. If the integer expression
 4787 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-
 4788 zero floating-point exception shall be raised.

4789 These functions may fail if:

4790 MX **Range Error** The value of x is subnormal.

4791 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 4792 then *errno* shall be set to [ERANGE]. If the integer expression
 4793 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
 4794 floating-point exception shall be raised.

4795 **EXAMPLES**

4796 None.

4797 **APPLICATION USAGE**

4798 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
 4799 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

4800 **RATIONALE**

4801 None.

4802 **FUTURE DIRECTIONS**

4803 None.

4804 **SEE ALSO**

4805 *feclearexcept()*, *fetestexcept()*, *tanh()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section
 4806 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

4807 **CHANGE HISTORY**

4808 First released in Issue 4, Version 2.

4809 **Issue 5**

4810 Moved from X/OPEN UNIX extension to BASE.

4811 **Issue 6**

4812 The *atanh()* function is no longer marked as an extension.

4813 The *atanhf()* and *atanhl()* functions are added for alignment with the ISO/IEC 9899:1999
 4814 standard.

4815 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
 4816 revised to align with the ISO/IEC 9899:1999 standard.

4817 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
 4818 marked.

4819 **NAME**

4820 atanl — arc tangent function

4821 **SYNOPSIS**

4822 #include <math.h>

4823 long double atanl(long double x);

4824 **DESCRIPTION**4825 Refer to *atan()*.

4826 **NAME**

4827 atexit — register a function to run at process termination

4828 **SYNOPSIS**

4829 #include <stdlib.h>

4830 int atexit(void (*func)(void));

4831 **DESCRIPTION**4832 CX The functionality described on this reference page is aligned with the ISO C standard. Any
4833 conflict between the requirements described here and the ISO C standard is unintentional. This
4834 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.4835 The *atexit()* function shall register the function pointed to by *func*, to be called without
4836 arguments at normal program termination. At normal program termination, all functions
4837 registered by the *atexit()* function shall be called, in the reverse order of their registration, except
4838 that a function is called after any previously registered functions that had already been called at
4839 the time it was registered. Normal termination occurs either by a call to *exit()* or a return from
4840 *main()*.4841 At least 32 functions can be registered with *atexit()*.4842 CX After a successful call to any of the *exec* functions, any functions previously registered by *atexit()*
4843 shall no longer be registered.4844 **RETURN VALUE**4845 Upon successful completion, *atexit()* shall return 0; otherwise, it shall return a non-zero value.4846 **ERRORS**

4847 No errors are defined.

4848 **EXAMPLES**

4849 None.

4850 **APPLICATION USAGE**4851 The functions registered by a call to *atexit()* must return to ensure that all registered functions
4852 are called.4853 The application should call *sysconf()* to obtain the value of {ATEXIT_MAX}, the number of
4854 functions that can be registered. There is no way for an application to tell how many functions
4855 have already been registered with *atexit()*.4856 **RATIONALE**

4857 None.

4858 **FUTURE DIRECTIONS**

4859 None.

4860 **SEE ALSO**4861 *exit()*, *sysconf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>4862 **CHANGE HISTORY**

4863 First released in Issue 4. Derived from the ANSI C standard.

4864 **Issue 6**

4865 Extensions beyond the ISO C standard are marked.

4866 The DESCRIPTION is updated for alignment with the ISO/IEC 9899:1999 standard.

4867 **NAME**4868 *atof* — convert a string to a double-precision number4869 **SYNOPSIS**4870 `#include <stdlib.h>`4871 `double atof(const char *str);`4872 **DESCRIPTION**

4873 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
4874 conflict between the requirements described here and the ISO C standard is unintentional. This
4875 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

4876 The call *atof(str)* shall be equivalent to:4877 `strtod(str, (char **)NULL),`

4878 except that the handling of errors may differ. If the value cannot be represented, the behavior is
4879 undefined.

4880 **RETURN VALUE**4881 The *atof()* function shall return the converted value if the value can be represented.4882 **ERRORS**

4883 No errors are defined.

4884 **EXAMPLES**

4885 None.

4886 **APPLICATION USAGE**

4887 The *atof()* function is subsumed by *strtod()* but is retained because it is used extensively in
4888 existing code. If the number is not known to be in range, *strtod()* should be used because *atof()* is
4889 not required to perform any error checking.

4890 **RATIONALE**

4891 None.

4892 **FUTURE DIRECTIONS**

4893 None.

4894 **SEE ALSO**4895 *strtod()*, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdlib.h>`4896 **CHANGE HISTORY**

4897 First released in Issue 1. Derived from Issue 1 of the SVID.

4898 **NAME**

4899 atoi — convert a string to an integer

4900 **SYNOPSIS**

4901 #include <stdlib.h>

4902 int atoi(const char *str);

4903 **DESCRIPTION**

4904 cx The functionality described on this reference page is aligned with the ISO C standard. Any
4905 conflict between the requirements described here and the ISO C standard is unintentional. This
4906 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

4907 The call *atoi(str)* shall be equivalent to:

4908 (int) strtol(str, (char **)NULL, 10)

4909 except that the handling of errors may differ. If the value cannot be represented, the behavior is
4910 undefined.

4911 **RETURN VALUE**4912 The *atoi()* function shall return the converted value if the value can be represented.4913 **ERRORS**

4914 No errors are defined.

4915 **EXAMPLES**4916 **Converting an Argument**

4917 The following example checks for proper usage of the program. If there is an argument and the
4918 decimal conversion of this argument (obtained using *atoi()*) is greater than 0, then the program
4919 has a valid number of minutes to wait for an event.

```
4920         #include <stdlib.h>
4921         #include <stdio.h>
4922         ...
4923         int minutes_to_event;
4924         ...
4925         if (argc < 2 || ((minutes_to_event = atoi (argv[1]))) <= 0) {
4926             fprintf(stderr, "Usage: %s minutes\n", argv[0]); exit(1);
4927         }
4928         ...
```

4929 **APPLICATION USAGE**

4930 The *atoi()* function is subsumed by *strtol()* but is retained because it is used extensively in
4931 existing code. If the number is not known to be in range, *strtol()* should be used because *atoi()* is
4932 not required to perform any error checking.

4933 **RATIONALE**

4934 None.

4935 **FUTURE DIRECTIONS**

4936 None.

4937 **SEE ALSO**4938 *strtol()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

4939 **CHANGE HISTORY**

4940 First released in Issue 1. Derived from Issue 1 of the SVID.

4941 NAME

4942 `atol`, `atoll` — convert a string to a long integer

4943 SYNOPSIS

4944 `#include <stdlib.h>`

4945 `long atol(const char *str);`

4946 `long long atoll(const char *nptr);`

4947 DESCRIPTION

4948 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
4949 conflict between the requirements described here and the ISO C standard is unintentional. This
4950 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

4951 The call `atol(str)` shall be equivalent to:

4952 `strtoul(str, (char **)NULL, 10)`

4953 The call `atoll(str)` shall be equivalent to:

4954 `strtoll(nptr, (char **)NULL, 10)`

4955 except that the handling of errors may differ. If the value cannot be represented, the behavior is
4956 undefined.

4957 RETURN VALUE

4958 These functions shall return the converted value if the value can be represented.

4959 ERRORS

4960 No errors are defined.

4961 EXAMPLES

4962 None.

4963 APPLICATION USAGE

4964 The `atol()` function is subsumed by `strtoul()` but is retained because it is used extensively in
4965 existing code. If the number is not known to be in range, `strtoul()` should be used because `atol()` is
4966 not required to perform any error checking.

4967 RATIONALE

4968 None.

4969 FUTURE DIRECTIONS

4970 None.

4971 SEE ALSO

4972 `strtoul()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdlib.h>`

4973 CHANGE HISTORY

4974 First released in Issue 1. Derived from Issue 1 of the SVID.

4975 Issue 6

4976 The `atoll()` function is added for alignment with the ISO/IEC 9899:1999 standard.

4977 **NAME**4978 **basename** — return the last component of a pathname4979 **SYNOPSIS**4980 XSI `#include <libgen.h>`4981 `char *basename(char *path);`

4982

4983 **DESCRIPTION**4984 The *basename()* function shall take the pathname pointed to by *path* and return a pointer to the
4985 final component of the pathname, deleting any trailing '/' characters.4986 If the string consists entirely of the '/' character, *basename()* shall return a pointer to the string
4987 "/". If the string is exactly "///", it is implementation-defined whether '/' or "///" is
4988 returned.4989 If *path* is a null pointer or points to an empty string, *basename()* shall return a pointer to the
4990 string ".".4991 The *basename()* function may modify the string pointed to by *path*, and may return a pointer to
4992 static storage that may then be overwritten by a subsequent call to *basename()*.4993 The *basename()* function need not be reentrant. A function that is not required to be reentrant is
4994 not required to be thread-safe.4995 **RETURN VALUE**4996 The *basename()* function shall return a pointer to the final component of *path*.4997 **ERRORS**

4998 No errors are defined.

4999 **EXAMPLES**5000 **Using basename()**5001 The following program fragment returns a pointer to the value *lib*, which is the base name of
5002 */usr/lib*.5003 `#include <libgen.h>`5004 `...`5005 `char *name = "/usr/lib";`5006 `char *base;`5007 `base = basename(name);`5008 `...`5009 **Sample Input and Output Strings for basename()**5010 In the following table, the input string is the value pointed to by *path*, and the output string is
5011 the return value of the *basename()* function.

Input String	Output String
"/usr/lib"	"lib"
"/usr/"	"usr"
"/"	"/"
"/" / "	"/"
"/usr//lib/"	"lib"

5018 APPLICATION USAGE

5019 None.

5020 RATIONALE

5021 None.

5022 FUTURE DIRECTIONS

5023 None.

5024 SEE ALSO

5025 *dirname()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<libgen.h>**, the Shell and
5026 Utilities volume of IEEE Std 1003.1-2001, *basename*

5027 CHANGE HISTORY

5028 First released in Issue 4, Version 2.

5029 Issue 5

5030 Moved from X/OPEN UNIX extension to BASE.

5031 Normative text previously in the APPLICATION USAGE section is moved to the
5032 DESCRIPTION.

5033 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

5034 Issue 6

5035 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

5036 **NAME**5037 bcmp — memory operations (**LEGACY**)5038 **SYNOPSIS**

5039 xSI #include <strings.h>

5040 int bcmp(const void *s1, const void *s2, size_t n);

5041

5042 **DESCRIPTION**5043 The *bcmp()* function shall compare the first *n* bytes of the area pointed to by *s1* with the area
5044 pointed to by *s2*.5045 **RETURN VALUE**5046 The *bcmp()* function shall return 0 if *s1* and *s2* are identical; otherwise, it shall return non-zero.
5047 Both areas are assumed to be *n* bytes long. If the value of *n* is 0, *bcmp()* shall return 0.5048 **ERRORS**

5049 No errors are defined.

5050 **EXAMPLES**

5051 None.

5052 **APPLICATION USAGE**5053 The *memcmp()* function is preferred over this function.5054 For maximum portability, it is recommended to replace the function call to *bcmp()* as follows:

5055 #define bcmp(b1,b2,len) memcmp((b1), (b2), (size_t)(len))

5056 **RATIONALE**

5057 None.

5058 **FUTURE DIRECTIONS**

5059 This function may be withdrawn in a future version.

5060 **SEE ALSO**5061 *memcmp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <strings.h>5062 **CHANGE HISTORY**

5063 First released in Issue 4, Version 2.

5064 **Issue 5**

5065 Moved from X/OPEN UNIX extension to BASE.

5066 **Issue 6**

5067 This function is marked LEGACY.

5068 **NAME**5069 bcopy — memory operations (**LEGACY**)5070 **SYNOPSIS**

5071 xSI #include <strings.h>

5072 void bcopy(const void *s1, void *s2, size_t n);

5073

5074 **DESCRIPTION**5075 The *bcopy()* function shall copy *n* bytes from the area pointed to by *s1* to the area pointed to by
5076 *s2*.5077 The bytes are copied correctly even if the area pointed to by *s1* overlaps the area pointed to by
5078 *s2*.5079 **RETURN VALUE**5080 The *bcopy()* function shall not return a value.5081 **ERRORS**

5082 No errors are defined.

5083 **EXAMPLES**

5084 None.

5085 **APPLICATION USAGE**5086 The *memmove()* function is preferred over this function.

5087 The following are approximately equivalent (note the order of the arguments):

5088 bcopy(s1,s2,n) ~ memmove(s2,s1,n)

5089 For maximum portability, it is recommended to replace the function call to *bcopy()* as follows:

5090 #define bcopy(b1,b2,len) (memmove((b2), (b1), (len)), (void) 0)

5091 **RATIONALE**

5092 None.

5093 **FUTURE DIRECTIONS**

5094 This function may be withdrawn in a future version.

5095 **SEE ALSO**5096 *memmove()*, the Base Definitions volume of IEEE Std 1003.1-2001, <strings.h>5097 **CHANGE HISTORY**

5098 First released in Issue 4, Version 2.

5099 **Issue 5**

5100 Moved from X/OPEN UNIX extension to BASE.

5101 **Issue 6**

5102 This function is marked LEGACY.

5103 **NAME**

5104 bind — bind a name to a socket

5105 **SYNOPSIS**

5106 #include <sys/socket.h>

```
5107       int bind(int socket, const struct sockaddr *address,
5108               socklen_t address_len);
```

5109 **DESCRIPTION**

5110 The *bind()* function shall assign a local socket address *address* to a socket identified by descriptor
 5111 *socket* that has no local socket address assigned. Sockets created with the *socket()* function are
 5112 initially unnamed; they are identified only by their address family.

5113 The *bind()* function takes the following arguments:

5114	<i>socket</i>	Specifies the file descriptor of the socket to be bound.
5115	<i>address</i>	Points to a sockaddr structure containing the address to be bound to the
5116		socket. The length and format of the address depend on the address family of
5117		the socket.
5118	<i>address_len</i>	Specifies the length of the sockaddr structure pointed to by the <i>address</i>
5119		argument.

5120 The socket specified by *socket* may require the process to have appropriate privileges to use the
 5121 *bind()* function.

5122 **RETURN VALUE**

5123 Upon successful completion, *bind()* shall return 0; otherwise, *-1* shall be returned and *errno* set
 5124 to indicate the error.

5125 **ERRORS**

5126 The *bind()* function shall fail if:

5127	[EADDRINUSE]	The specified address is already in use.
5128	[EADDRNOTAVAIL]	
5129		The specified address is not available from the local machine.
5130	[EAFNOSUPPORT]	
5131		The specified address is not a valid address for the address family of the
5132		specified socket.
5133	[EBADF]	The <i>socket</i> argument is not a valid file descriptor.
5134	[EINVAL]	The socket is already bound to an address, and the protocol does not support
5135		binding to a new address; or the socket has been shut down.
5136	[ENOTSOCK]	The <i>socket</i> argument does not refer to a socket.
5137	[EOPNOTSUPP]	The socket type of the specified socket does not support binding to an
5138		address.
5139		If the address family of the socket is AF_UNIX, then <i>bind()</i> shall fail if:
5140	[EACCES]	A component of the path prefix denies search permission, or the requested
5141		name requires writing in a directory with a mode that denies write
5142		permission.
5143	[EDESTADDRREQ] or [EISDIR]	
5144		The <i>address</i> argument is a null pointer.

5145	[EIO]	An I/O error occurred.
5146	[ELOOP]	A loop exists in symbolic links encountered during resolution of the pathname in <i>address</i> .
5147		
5148	[ENAMETOOLONG]	
5149		A component of a pathname exceeded {NAME_MAX} characters, or an entire
5150		pathname exceeded {PATH_MAX} characters.
5151	[ENOENT]	A component of the pathname does not name an existing file or the pathname
5152		is an empty string.
5153	[ENOTDIR]	A component of the path prefix of the pathname in <i>address</i> is not a directory.
5154	[EROFS]	The name would reside on a read-only file system.
5155		The <i>bind()</i> function may fail if:
5156	[EACCES]	The specified address is protected and the current user does not have
5157		permission to bind to it.
5158	[EINVAL]	The <i>address_len</i> argument is not a valid length for the address family.
5159	[EISCONN]	The socket is already connected.
5160	[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during
5161		resolution of the pathname in <i>address</i> .
5162	[ENAMETOOLONG]	
5163		Pathname resolution of a symbolic link produced an intermediate result
5164		whose length exceeds {PATH_MAX}.
5165	[ENOBUFS]	Insufficient resources were available to complete the call.

5166 EXAMPLES

5167 None.

5168 APPLICATION USAGE

5169 An application program can retrieve the assigned socket name with the *getsockname()* function.

5170 RATIONALE

5171 None.

5172 FUTURE DIRECTIONS

5173 None.

5174 SEE ALSO

5175 *connect()*, *getsockname()*, *listen()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-2001,
5176 <sys/socket.h>

5177 CHANGE HISTORY

5178 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

5179 **NAME**5180 `bsd_signal` — simplified signal facilities5181 **SYNOPSIS**5182 OB XSI `#include <signal.h>`5183 `void (*bsd_signal(int sig, void (*func)(int)))(int);`

5184

5185 **DESCRIPTION**5186 The `bsd_signal()` function provides a partially compatible interface for programs written to
5187 historical system interfaces (see APPLICATION USAGE).5188 The function call `bsd_signal(sig, func)` shall be equivalent to the following:5189 `void (*bsd_signal(int sig, void (*func)(int)))(int)`

5190 {

5191 `struct sigaction act, oact;`5192 `act.sa_handler = func;`5193 `act.sa_flags = SA_RESTART;`5194 `sigemptyset(&act.sa_mask);`5195 `sigaddset(&act.sa_mask, sig);`5196 `if (sigaction(sig, &act, &oact) == -1)`5197 `return(SIG_ERR);`5198 `return(oact.sa_handler);`

5199 }

5200 The handler function should be declared:

5201 `void handler(int sig);`5202 where *sig* is the signal number. The behavior is undefined if *func* is a function that takes more
5203 than one argument, or an argument of a different type.5204 **RETURN VALUE**5205 Upon successful completion, `bsd_signal()` shall return the previous action for *sig*. Otherwise,
5206 `SIG_ERR` shall be returned and *errno* shall be set to indicate the error.5207 **ERRORS**5208 Refer to `sigaction()`.5209 **EXAMPLES**

5210 None.

5211 **APPLICATION USAGE**5212 This function is a direct replacement for the BSD `signal()` function for simple applications that
5213 are installing a single-argument signal handler function. If a BSD signal handler function is being
5214 installed that expects more than one argument, the application has to be modified to use
5215 `sigaction()`. The `bsd_signal()` function differs from `signal()` in that the `SA_RESTART` flag is set
5216 and the `SA_RESETHAND` is clear when `bsd_signal()` is used. The state of these flags is not
5217 specified for `signal()`.5218 It is recommended that new applications use the `sigaction()` function.5219 **RATIONALE**

5220 None.

5221 **FUTURE DIRECTIONS**

5222 None.

5223 **SEE ALSO**

5224 *sigaction()*, *sigaddset()*, *sigemptyset()*, *signal()*, the Base Definitions volume of
5225 IEEE Std 1003.1-2001, <signal.h>

5226 **CHANGE HISTORY**

5227 First released in Issue 4, Version 2.

5228 **Issue 5**

5229 Moved from X/OPEN UNIX extension to BASE.

5230 **Issue 6**

5231 This function is marked obsolescent.

5232 **NAME**

5233 bsearch — binary search a sorted table

5234 **SYNOPSIS**

5235 #include <stdlib.h>

```
5236       void *bsearch(const void *key, const void *base, size_t nel,
5237                   size_t width, int (*compar)(const void *, const void *));
```

5238 **DESCRIPTION**

5239 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 5240 conflict between the requirements described here and the ISO C standard is unintentional. This
 5241 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5242 The *bsearch()* function shall search an array of *nel* objects, the initial element of which is pointed
 5243 to by *base*, for an element that matches the object pointed to by *key*. The size of each element in
 5244 the array is specified by *width*.

5245 The comparison function pointed to by *compar* shall be called with two arguments that point to
 5246 the *key* object and to an array element, in that order.

5247 The application shall ensure that the function returns an integer less than, equal to, or greater
 5248 than 0 if the *key* object is considered, respectively, to be less than, to match, or to be greater than
 5249 the array element. The application shall ensure that the array consists of all the elements that
 5250 compare less than, all the elements that compare equal to, and all the elements that compare
 5251 greater than the *key* object, in that order.

5252 **RETURN VALUE**

5253 The *bsearch()* function shall return a pointer to a matching member of the array, or a null pointer
 5254 if no match is found. If two or more members compare equal, which member is returned is
 5255 unspecified.

5256 **ERRORS**

5257 No errors are defined.

5258 **EXAMPLES**

5259 The example below searches a table containing pointers to nodes consisting of a string and its
 5260 length. The table is ordered alphabetically on the string in the node pointed to by each entry.

5261 The code fragment below reads in strings and either finds the corresponding node and prints out
 5262 the string and its length, or prints an error message.

```
5263       #include <stdio.h>
5264       #include <stdlib.h>
5265       #include <string.h>

5266       #define TABSIZE     1000

5267       struct node {                                 /* These are stored in the table. */
5268           char *string;
5269           int length;
5270       };
5271       struct node table[TABSIZE];     /* Table to be searched. */
5272       .
5273       .
5274       .
5275       {
5276           struct node *node_ptr, node;
5277           /* Routine to compare 2 nodes. */
```



```

5278     int node_compare(const void *, const void *);
5279     char str_space[20];    /* Space to read string into. */
5280     .
5281     .
5282     .
5283     node.string = str_space;
5284     while (scanf("%s", node.string) != EOF) {
5285         node_ptr = (struct node *)bsearch((void *)&node,
5286             (void *)table, TABSIZE,
5287             sizeof(struct node), node_compare);
5288         if (node_ptr != NULL) {
5289             (void)printf("string = %20s, length = %d\n",
5290                 node_ptr->string, node_ptr->length);
5291         } else {
5292             (void)printf("not found: %s\n", node.string);
5293         }
5294     }
5295 }
5296 /*
5297     This routine compares two nodes based on an
5298     alphabetical ordering of the string field.
5299 */
5300 int
5301 node_compare(const void *node1, const void *node2)
5302 {
5303     return strcmp(((const struct node *)node1)->string,
5304         ((const struct node *)node2)->string);
5305 }

```

5306 APPLICATION USAGE

5307 The pointers to the key and the element at the base of the table should be of type pointer-to-
5308 element.

5309 The comparison function need not compare every byte, so arbitrary data may be contained in
5310 the elements in addition to the values being compared.

5311 In practice, the array is usually sorted according to the comparison function.

5312 RATIONALE

5313 None.

5314 FUTURE DIRECTIONS

5315 None.

5316 SEE ALSO

5317 *bcreate()*, *bsearch()*, *qsort()*, *tsearch()*, the Base Definitions volume of IEEE Std 1003.1-2001,
5318 **<stdlib.h>**

5319 CHANGE HISTORY

5320 First released in Issue 1. Derived from Issue 1 of the SVID.

5321 Issue 6

5322 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

5323 **NAME**

5324 btowc — single byte to wide character conversion

5325 **SYNOPSIS**

5326 #include <stdio.h>

5327 #include <wchar.h>

5328 wint_t btowc(int c);

5329 **DESCRIPTION**

5330 cx The functionality described on this reference page is aligned with the ISO C standard. Any
5331 conflict between the requirements described here and the ISO C standard is unintentional. This
5332 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5333 The *btowc()* function shall determine whether *c* constitutes a valid (one-byte) character in the
5334 initial shift state.

5335 The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

5336 **RETURN VALUE**

5337 The *btowc()* function shall return WEOF if *c* has the value EOF or if (**unsigned char**) *c* does not
5338 constitute a valid (one-byte) character in the initial shift state. Otherwise, it shall return the
5339 wide-character representation of that character.

5340 **ERRORS**

5341 No errors are defined.

5342 **EXAMPLES**

5343 None.

5344 **APPLICATION USAGE**

5345 None.

5346 **RATIONALE**

5347 None.

5348 **FUTURE DIRECTIONS**

5349 None.

5350 **SEE ALSO**5351 *wctob()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>5352 **CHANGE HISTORY**

5353 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
5354 (E).

5355 **NAME**5356 **bzero** — memory operations (**LEGACY**)5357 **SYNOPSIS**

5358 XSI #include <strings.h>

5359 void bzero(void *s, size_t n);

5360

5361 **DESCRIPTION**5362 The *bzero()* function shall place *n* zero-valued bytes in the area pointed to by *s*.5363 **RETURN VALUE**5364 The *bzero()* function shall not return a value.5365 **ERRORS**

5366 No errors are defined.

5367 **EXAMPLES**

5368 None.

5369 **APPLICATION USAGE**5370 The *memset()* function is preferred over this function.5371 For maximum portability, it is recommended to replace the function call to *bzero()* as follows:

5372 #define bzero(b,len) (memset((b), '\0', (len)), (void) 0)

5373 **RATIONALE**

5374 None.

5375 **FUTURE DIRECTIONS**

5376 This function may be withdrawn in a future version.

5377 **SEE ALSO**5378 *memset()*, the Base Definitions volume of IEEE Std 1003.1-2001, <strings.h>5379 **CHANGE HISTORY**

5380 First released in Issue 4, Version 2.

5381 **Issue 5**

5382 Moved from X/OPEN UNIX extension to BASE.

5383 **Issue 6**

5384 This function is marked LEGACY.

5385 **NAME**

5386 cabs, cabsf, cabsl — return a complex absolute value

5387 **SYNOPSIS**

5388 #include <complex.h>

5389 double cabs(double complex z);

5390 float cabsf(float complex z);

5391 long double cabsl(long double complex z);

5392 **DESCRIPTION**

5393 cx The functionality described on this reference page is aligned with the ISO C standard. Any
5394 conflict between the requirements described here and the ISO C standard is unintentional. This
5395 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5396 These functions shall compute the complex absolute value (also called norm, modulus, or
5397 magnitude) of z.

5398 **RETURN VALUE**

5399 These functions shall return the complex absolute value.

5400 **ERRORS**

5401 No errors are defined.

5402 **EXAMPLES**

5403 None.

5404 **APPLICATION USAGE**

5405 None.

5406 **RATIONALE**

5407 None.

5408 **FUTURE DIRECTIONS**

5409 None.

5410 **SEE ALSO**

5411 The Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

5412 **CHANGE HISTORY**

5413 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

5414 **NAME**

5415 cacos, cacosf, cacosl — complex arc cosine functions

5416 **SYNOPSIS**

5417 #include <complex.h>

5418 double complex cacos(double complex z);

5419 float complex cacosf(float complex z);

5420 long double complex cacosl(long double complex z);

5421 **DESCRIPTION**

5422 cx The functionality described on this reference page is aligned with the ISO C standard. Any
5423 conflict between the requirements described here and the ISO C standard is unintentional. This
5424 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5425 These functions shall compute the complex arc cosine of z , with branch cuts outside the interval
5426 $[-1, +1]$ along the real axis.

5427 **RETURN VALUE**

5428 These functions shall return the complex arc cosine value, in the range of a strip mathematically
5429 unbounded along the imaginary axis and in the interval $[0, \pi]$ along the real axis.

5430 **ERRORS**

5431 No errors are defined.

5432 **EXAMPLES**

5433 None.

5434 **APPLICATION USAGE**

5435 None.

5436 **RATIONALE**

5437 None.

5438 **FUTURE DIRECTIONS**

5439 None.

5440 **SEE ALSO**

5441 ccos(), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

5442 **CHANGE HISTORY**

5443 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

5444 **NAME**

5445 cacosh, cacoshf, cacoshl — complex arc hyperbolic cosine functions

5446 **SYNOPSIS**

5447 #include <complex.h>

5448 double complex cacosh(double complex z);

5449 float complex cacoshf(float complex z);

5450 long double complex cacoshl(long double complex z);

5451 **DESCRIPTION**

5452 cx The functionality described on this reference page is aligned with the ISO C standard. Any
5453 conflict between the requirements described here and the ISO C standard is unintentional. This
5454 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5455 These functions shall compute the complex arc hyperbolic cosine of *z*, with a branch cut at
5456 values less than 1 along the real axis.

5457 **RETURN VALUE**

5458 These functions shall return the complex arc hyperbolic cosine value, in the range of a half-strip
5459 of non-negative values along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary axis.

5460 **ERRORS**

5461 No errors are defined.

5462 **EXAMPLES**

5463 None.

5464 **APPLICATION USAGE**

5465 None.

5466 **RATIONALE**

5467 None.

5468 **FUTURE DIRECTIONS**

5469 None.

5470 **SEE ALSO**

5471 ccosh(), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

5472 **CHANGE HISTORY**

5473 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

5474 **NAME**

5475 cacosl — complex arc cosine functions

5476 **SYNOPSIS**

5477 #include <complex.h>

5478 long double complex cacosl(long double complex *z*);5479 **DESCRIPTION**5480 Refer to *cacos()*.

5481 **NAME**

5482 calloc — a memory allocator

5483 **SYNOPSIS**

5484 #include <stdlib.h>

5485 void *calloc(size_t *nelem*, size_t *elsize*);5486 **DESCRIPTION**

5487 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 5488 conflict between the requirements described here and the ISO C standard is unintentional. This
 5489 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5490 The *calloc()* function shall allocate unused space for an array of *nelem* elements each of whose
 5491 size in bytes is *elsize*. The space shall be initialized to all bits 0.

5492 The order and contiguity of storage allocated by successive calls to *calloc()* is unspecified. The
 5493 pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to
 5494 a pointer to any type of object and then used to access such an object or an array of such objects
 5495 in the space allocated (until the space is explicitly freed or reallocated). Each such allocation
 5496 shall yield a pointer to an object disjoint from any other object. The pointer returned shall point
 5497 to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null
 5498 pointer shall be returned. If the size of the space requested is 0, the behavior is implementation-
 5499 defined: the value returned shall be either a null pointer or a unique pointer.

5500 **RETURN VALUE**

5501 Upon successful completion with both *nelem* and *elsize* non-zero, *calloc()* shall return a pointer to
 5502 the allocated space. If either *nelem* or *elsize* is 0, then either a null pointer or a unique pointer
 5503 value that can be successfully passed to *free()* shall be returned. Otherwise, it shall return a null
 5504 CX pointer and set *errno* to indicate the error.

5505 **ERRORS**5506 The *calloc()* function shall fail if:

5507 CX [ENOMEM] Insufficient memory is available.

5508 **EXAMPLES**

5509 None.

5510 **APPLICATION USAGE**

5511 There is now no requirement for the implementation to support the inclusion of <malloc.h>.

5512 **RATIONALE**

5513 None.

5514 **FUTURE DIRECTIONS**

5515 None.

5516 **SEE ALSO**5517 *free()*, *malloc()*, *realloc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>5518 **CHANGE HISTORY**

5519 First released in Issue 1. Derived from Issue 1 of the SVID.

5520 **Issue 6**

5521 Extensions beyond the ISO C standard are marked.

- 5522 The following new requirements on POSIX implementations derive from alignment with the
5523 Single UNIX Specification:
- 5524 • The setting of *errno* and the [ENOMEM] error condition are mandatory if an insufficient
5525 memory condition occurs.

5526 **NAME**

5527 carg, cargf, cargl — complex argument functions

5528 **SYNOPSIS**

5529 #include <complex.h>

5530 double carg(double complex z);

5531 float cargf(float complex z);

5532 long double cargl(long double complex z);

5533 **DESCRIPTION**

5534 cx The functionality described on this reference page is aligned with the ISO C standard. Any
5535 conflict between the requirements described here and the ISO C standard is unintentional. This
5536 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5537 These functions shall compute the argument (also called phase angle) of *z*, with a branch cut
5538 along the negative real axis.

5539 **RETURN VALUE**5540 These functions shall return the value of the argument in the interval $[-\pi, +\pi]$.5541 **ERRORS**

5542 No errors are defined.

5543 **EXAMPLES**

5544 None.

5545 **APPLICATION USAGE**

5546 None.

5547 **RATIONALE**

5548 None.

5549 **FUTURE DIRECTIONS**

5550 None.

5551 **SEE ALSO**5552 *cimag()*, *conj()*, *cproj()*, the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>5553 **CHANGE HISTORY**

5554 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

5555 NAME

5556 `casin`, `casinf`, `casinl` — complex arc sine functions

5557 SYNOPSIS

5558 `#include <complex.h>`

5559 `double complex casin(double complex z);`

5560 `float complex casinf(float complex z);`

5561 `long double complex casinl(long double complex z);`

5562 DESCRIPTION

5563 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
5564 conflict between the requirements described here and the ISO C standard is unintentional. This
5565 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5566 These functions shall compute the complex arc sine of *z*, with branch cuts outside the interval
5567 $[-1, +1]$ along the real axis.

5568 RETURN VALUE

5569 These functions shall return the complex arc sine value, in the range of a strip mathematically
5570 unbounded along the imaginary axis and in the interval $[-\pi/2, +\pi/2]$ along the real axis.

5571 ERRORS

5572 No errors are defined.

5573 EXAMPLES

5574 None.

5575 APPLICATION USAGE

5576 None.

5577 RATIONALE

5578 None.

5579 FUTURE DIRECTIONS

5580 None.

5581 SEE ALSO

5582 `csin()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<complex.h>`

5583 CHANGE HISTORY

5584 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

5585 **NAME**

5586 casinh, casinhf, casinhl — complex arc hyperbolic sine functions

5587 **SYNOPSIS**

5588 #include <complex.h>

5589 double complex casinh(double complex z);

5590 float complex casinhf(float complex z);

5591 long double complex casinhl(long double complex z);

5592 **DESCRIPTION**

5593 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
5594 conflict between the requirements described here and the ISO C standard is unintentional. This
5595 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5596 These functions shall compute the complex arc hyperbolic sine of z , with branch cuts outside the
5597 interval $[-i, +i]$ along the imaginary axis.

5598 **RETURN VALUE**

5599 These functions shall return the complex arc hyperbolic sine value, in the range of a strip
5600 mathematically unbounded along the real axis and in the interval $[-i\pi/2, +i\pi/2]$ along the
5601 imaginary axis.

5602 **ERRORS**

5603 No errors are defined.

5604 **EXAMPLES**

5605 None.

5606 **APPLICATION USAGE**

5607 None.

5608 **RATIONALE**

5609 None.

5610 **FUTURE DIRECTIONS**

5611 None.

5612 **SEE ALSO**5613 `csinh()`, the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>5614 **CHANGE HISTORY**

5615 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

5616 **NAME**5617 **casinl** — complex arc sine functions5618 **SYNOPSIS**5619 `#include <complex.h>`5620 `long double complex casinl(long double complex z);`5621 **DESCRIPTION**5622 Refer to *casin()*.

5623 **NAME**

5624 catan, catanf, catanl — complex arc tangent functions

5625 **SYNOPSIS**

5626 #include <complex.h>

5627 double complex catan(double complex *z*);5628 float complex catanf(float complex *z*);5629 long double complex catanl(long double complex *z*);5630 **DESCRIPTION**

5631 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
5632 conflict between the requirements described here and the ISO C standard is unintentional. This
5633 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5634 These functions shall compute the complex arc tangent of *z*, with branch cuts outside the
5635 interval $[-i, +i]$ along the imaginary axis.

5636 **RETURN VALUE**

5637 These functions shall return the complex arc tangent value, in the range of a strip
5638 mathematically unbounded along the imaginary axis and in the interval $[-\pi/2, +\pi/2]$ along the
5639 real axis.

5640 **ERRORS**

5641 No errors are defined.

5642 **EXAMPLES**

5643 None.

5644 **APPLICATION USAGE**

5645 None.

5646 **RATIONALE**

5647 None.

5648 **FUTURE DIRECTIONS**

5649 None.

5650 **SEE ALSO**

5651 ctan(), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

5652 **CHANGE HISTORY**

5653 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

5654 **NAME**

5655 catanh, catanhf, catanhl — complex arc hyperbolic tangent functions

5656 **SYNOPSIS**

5657 #include <complex.h>

5658 double complex catanh(double complex z);

5659 float complex catanhf(float complex z);

5660 long double complex catanhl(long double complex z);

5661 **DESCRIPTION**

5662 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
5663 conflict between the requirements described here and the ISO C standard is unintentional. This
5664 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5665 These functions shall compute the complex arc hyperbolic tangent of z , with branch cuts outside
5666 the interval $[-1, +1]$ along the real axis.

5667 **RETURN VALUE**

5668 These functions shall return the complex arc hyperbolic tangent value, in the range of a strip
5669 mathematically unbounded along the real axis and in the interval $[-i\pi/2, +i\pi/2]$ along the
5670 imaginary axis.

5671 **ERRORS**

5672 No errors are defined.

5673 **EXAMPLES**

5674 None.

5675 **APPLICATION USAGE**

5676 None.

5677 **RATIONALE**

5678 None.

5679 **FUTURE DIRECTIONS**

5680 None.

5681 **SEE ALSO**5682 `ctanh()`, the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>5683 **CHANGE HISTORY**

5684 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

5685 **NAME**

5686 catanl — complex arc tangent functions

5687 **SYNOPSIS**

5688 #include <complex.h>

5689 long double complex catanl(long double complex *z*);5690 **DESCRIPTION**5691 Refer to *catan()*.

5692 **NAME**

5693 catclose — close a message catalog descriptor

5694 **SYNOPSIS**

5695 XSI #include <nl_types.h>

5696 int catclose(nl_catd catd);

5697

5698 **DESCRIPTION**5699 The *catclose()* function shall close the message catalog identified by *catd*. If a file descriptor is
5700 used to implement the type **nl_catd**, that file descriptor shall be closed.5701 **RETURN VALUE**5702 Upon successful completion, *catclose()* shall return 0; otherwise, -1 shall be returned, and *errno*
5703 set to indicate the error.5704 **ERRORS**5705 The *catclose()* function may fail if:

5706 [EBADF] The catalog descriptor is not valid.

5707 [EINTR] The *catclose()* function was interrupted by a signal.5708 **EXAMPLES**

5709 None.

5710 **APPLICATION USAGE**

5711 None.

5712 **RATIONALE**

5713 None.

5714 **FUTURE DIRECTIONS**

5715 None.

5716 **SEE ALSO**5717 *catgets()*, *catopen()*, the Base Definitions volume of IEEE Std 1003.1-2001, <nl_types.h>5718 **CHANGE HISTORY**

5719 First released in Issue 2.

5720 **NAME**

5721 catgets — read a program message

5722 **SYNOPSIS**

5723 XSI #include <nl_types.h>

5724 char *catgets(nl_catd catd, int set_id, int msg_id, const char *s);

5725

5726 **DESCRIPTION**

5727 The *catgets()* function shall attempt to read message *msg_id*, in set *set_id*, from the message catalog identified by *catd*. The *catd* argument is a message catalog descriptor returned from an earlier call to *catopen()*. The *s* argument points to a default message string which shall be returned by *catgets()* if it cannot retrieve the identified message.

5731 The *catgets()* function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

5733 **RETURN VALUE**

5734 If the identified message is retrieved successfully, *catgets()* shall return a pointer to an internal buffer area containing the null-terminated message string. If the call is unsuccessful for any reason, *s* shall be returned and *errno* may be set to indicate the error.

5737 **ERRORS**5738 The *catgets()* function may fail if:

- | | | |
|------|-----------|--|
| 5739 | [EBADF] | The <i>catd</i> argument is not a valid message catalog descriptor open for reading. |
| 5740 | [EBADMSG] | The message identified by <i>set_id</i> and <i>msg_id</i> in the specified message catalog did not satisfy implementation-defined security criteria. |
| 5741 | | |
| 5742 | [EINTR] | The read operation was terminated due to the receipt of a signal, and no data was transferred. |
| 5743 | | |
| 5744 | [EINVAL] | The message catalog identified by <i>catd</i> is corrupted. |
| 5745 | [ENOMSG] | The message identified by <i>set_id</i> and <i>msg_id</i> is not in the message catalog. |

5746 **EXAMPLES**

5747 None.

5748 **APPLICATION USAGE**

5749 None.

5750 **RATIONALE**

5751 None.

5752 **FUTURE DIRECTIONS**

5753 None.

5754 **SEE ALSO**5755 *catclose()*, *catopen()*, the Base Definitions volume of IEEE Std 1003.1-2001, <nl_types.h>5756 **CHANGE HISTORY**

5757 First released in Issue 2.

5758 **Issue 5**

5759 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

Issue 6

5761 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

5762 **NAME**

5763 catopen — open a message catalog

5764 **SYNOPSIS**

5765 XSI #include <nl_types.h>

5766 nl_catd catopen(const char *name, int oflag);

5767

5768 **DESCRIPTION**

5769 The *catopen()* function shall open a message catalog and return a message catalog descriptor.
 5770 The *name* argument specifies the name of the message catalog to be opened. If *name* contains a
 5771 ' / ', then *name* specifies a complete name for the message catalog. Otherwise, the environment
 5772 variable *NLSPATH* is used with *name* substituted for the %N conversion specification (see the
 5773 Base Definitions volume of IEEE Std 1003.1-2001, Chapter 8, Environment Variables). If
 5774 *NLSPATH* exists in the environment when the process starts, then if the process has appropriate
 5775 privileges, the behavior of *catopen()* is undefined. If *NLSPATH* does not exist in the environment,
 5776 or if a message catalog cannot be found in any of the components specified by *NLSPATH*, then
 5777 an implementation-defined default path shall be used. This default may be affected by the
 5778 setting of *LC_MESSAGES* if the value of *oflag* is *NL_CAT_LOCALE*, or the *LANG* environment
 5779 variable if *oflag* is 0.

5780 A message catalog descriptor shall remain valid in a process until that process closes it, or a
 5781 successful call to one of the *exec* functions. A change in the setting of the *LC_MESSAGES*
 5782 category may invalidate existing open catalogs.

5783 If a file descriptor is used to implement message catalog descriptors, the *FD_CLOEXEC* flag
 5784 shall be set; see <fcntl.h>.

5785 If the value of the *oflag* argument is 0, the *LANG* environment variable is used to locate the
 5786 catalog without regard to the *LC_MESSAGES* category. If the *oflag* argument is
 5787 *NL_CAT_LOCALE*, the *LC_MESSAGES* category is used to locate the message catalog (see the
 5788 Base Definitions volume of IEEE Std 1003.1-2001, Section 8.2, Internationalization Variables).

5789 **RETURN VALUE**

5790 Upon successful completion, *catopen()* shall return a message catalog descriptor for use on
 5791 subsequent calls to *catgets()* and *catclose()*. Otherwise, *catopen()* shall return (*nl_catd*) -1 and set
 5792 *errno* to indicate the error.

5793 **ERRORS**5794 The *catopen()* function may fail if:

5795 [EACCES] Search permission is denied for the component of the path prefix of the
 5796 message catalog or read permission is denied for the message catalog.

5797 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

5798 [ENAMETOOLONG]

5799 The length of a pathname of the message catalog exceeds {PATH_MAX} or a
 5800 pathname component is longer than {NAME_MAX}.

5801 [ENAMETOOLONG]

5802 Pathname resolution of a symbolic link produced an intermediate result
 5803 whose length exceeds {PATH_MAX}.

5804 [ENFILE] Too many files are currently open in the system.

5805 [ENOENT] The message catalog does not exist or the *name* argument points to an empty
 5806 string.

5807 [ENOMEM] Insufficient storage space is available.
5808 [ENOTDIR] A component of the path prefix of the message catalog is not a directory.

5809 EXAMPLES

5810 None.

5811 APPLICATION USAGE

5812 Some implementations of *catopen()* use *malloc()* to allocate space for internal buffer areas. The
5813 *catopen()* function may fail if there is insufficient storage space available to accommodate these
5814 buffers.

5815 Conforming applications must assume that message catalog descriptors are not valid after a call
5816 to one of the *exec* functions.

5817 Application writers should be aware that guidelines for the location of message catalogs have
5818 not yet been developed. Therefore they should take care to avoid conflicting with catalogs used
5819 by other applications and the standard utilities.

5820 RATIONALE

5821 None.

5822 FUTURE DIRECTIONS

5823 None.

5824 SEE ALSO

5825 *catclose()*, *catgets()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<fcntl.h>**,
5826 **<nl_types.h>**, the Shell and Utilities volume of IEEE Std 1003.1-2001

5827 CHANGE HISTORY

5828 First released in Issue 2.

5829 **NAME**

5830 cbrt, cbrtf, cbrtl — cube root functions

5831 **SYNOPSIS**

5832 #include <math.h>

5833 double cbrt(double x);

5834 float cbrtf(float x);

5835 long double cbrtl(long double x);

5836 **DESCRIPTION**

5837 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 5838 conflict between the requirements described here and the ISO C standard is unintentional. This
 5839 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5840 These functions shall compute the real cube root of their argument *x*.5841 **RETURN VALUE**5842 Upon successful completion, these functions shall return the cube root of *x*.5843 **MX** If *x* is NaN, a NaN shall be returned.5844 If *x* is ± 0 or $\pm \text{Inf}$, *x* shall be returned.5845 **ERRORS**

5846 No errors are defined.

5847 **EXAMPLES**

5848 None.

5849 **APPLICATION USAGE**

5850 None.

5851 **RATIONALE**

5852 For some applications, a true cube root function, which returns negative results for negative
 5853 arguments, is more appropriate than *pow(x, 1.0/3.0)*, which returns a NaN for *x* less than 0.

5854 **FUTURE DIRECTIONS**

5855 None.

5856 **SEE ALSO**

5857 The Base Definitions volume of IEEE Std 1003.1-2001, <math.h>

5858 **CHANGE HISTORY**

5859 First released in Issue 4, Version 2.

5860 **Issue 5**

5861 Moved from X/OPEN UNIX extension to BASE.

5862 **Issue 6**5863 The *cbrt()* function is no longer marked as an extension.5864 The *cbrtf()* and *cbrtl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

5865 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
 5866 revised to align with the ISO/IEC 9899:1999 standard.

5867 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
 5868 marked.

5869 **NAME**

5870 ccos, ccosh, ccosl — complex cosine functions

5871 **SYNOPSIS**

5872 #include <complex.h>

5873 double complex ccos(double complex z);

5874 float complex ccosh(float complex z);

5875 long double complex ccosl(long double complex z);

5876 **DESCRIPTION**

5877 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5878 conflict between the requirements described here and the ISO C standard is unintentional. This
5879 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5880 These functions shall compute the complex cosine of z.

5881 **RETURN VALUE**

5882 These functions shall return the complex cosine value.

5883 **ERRORS**

5884 No errors are defined.

5885 **EXAMPLES**

5886 None.

5887 **APPLICATION USAGE**

5888 None.

5889 **RATIONALE**

5890 None.

5891 **FUTURE DIRECTIONS**

5892 None.

5893 **SEE ALSO**5894 *ccosh()*, the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>5895 **CHANGE HISTORY**

5896 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

5897 **NAME**

5898 ccosh, ccoshf, ccoshl — complex hyperbolic cosine functions

5899 **SYNOPSIS**

5900 #include <complex.h>

5901 double complex ccosh(double complex z);

5902 float complex ccoshf(float complex z);

5903 long double complex ccoshl(long double complex z);

5904 **DESCRIPTION**

5905 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
5906 conflict between the requirements described here and the ISO C standard is unintentional. This
5907 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5908 These functions shall compute the complex hyperbolic cosine of *z*.5909 **RETURN VALUE**

5910 These functions shall return the complex hyperbolic cosine value.

5911 **ERRORS**

5912 No errors are defined.

5913 **EXAMPLES**

5914 None.

5915 **APPLICATION USAGE**

5916 None.

5917 **RATIONALE**

5918 None.

5919 **FUTURE DIRECTIONS**

5920 None.

5921 **SEE ALSO**5922 *cacosh()*, the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>5923 **CHANGE HISTORY**

5924 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

5925 **NAME**

5926 ccosl — complex cosine functions

5927 **SYNOPSIS**

5928 #include <complex.h>

5929 long double complex ccosl(long double complex *z*);5930 **DESCRIPTION**5931 Refer to *ccos()*.

5932 **NAME**

5933 ceil, ceilf, ceill — ceiling value function

5934 **SYNOPSIS**

5935 #include <math.h>

5936 double ceil(double x);

5937 float ceilf(float x);

5938 long double ceill(long double x);

5939 **DESCRIPTION**

5940 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 5941 conflict between the requirements described here and the ISO C standard is unintentional. This
 5942 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5943 These functions shall compute the smallest integral value not less than *x*.

5944 An application wishing to check for error situations should set *errno* to zero and call
 5945 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 5946 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 5947 zero, an error has occurred.

5948 **RETURN VALUE**

5949 Upon successful completion, *ceil()*, *ceilf()*, and *ceill()* shall return the smallest integral value not
 5950 less than *x*, expressed as a type **double**, **float**, or **long double**, respectively.

5951 **MX** If *x* is NaN, a NaN shall be returned.5952 If *x* is ± 0 or $\pm \text{Inf}$, *x* shall be returned.

5953 **XSI** If the correct value would cause overflow, a range error shall occur and *ceil()*, *ceilf()*, and *ceill()*
 5954 shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

5955 **ERRORS**

5956 These functions shall fail if:

5957 **XSI** Range Error The result overflows.

5958 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 5959 then *errno* shall be set to [ERANGE]. If the integer expression
 5960 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow
 5961 floating-point exception shall be raised.

5962 **EXAMPLES**

5963 None.

5964 **APPLICATION USAGE**

5965 The integral value returned by these functions need not be expressible as an **int** or **long**. The
 5966 return value should be tested before assigning it to an integer type to avoid the undefined results
 5967 of an integer overflow.

5968 The *ceil()* function can only overflow when the floating-point representation has
 5969 DBL_MANT_DIG > DBL_MAX_EXP.

5970 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
 5971 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

5972 **RATIONALE**

5973 None.

5974 **FUTURE DIRECTIONS**

5975 None.

5976 **SEE ALSO**5977 *feclearexcept()*, *fetestexcept()*, *floor()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-2001,
5978 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>5979 **CHANGE HISTORY**

5980 First released in Issue 1. Derived from Issue 1 of the SVID.

5981 **Issue 5**5982 The DESCRIPTION is updated to indicate how an application should check for an error. This
5983 text was previously published in the APPLICATION USAGE section.5984 **Issue 6**5985 The *ceilf()* and *ceilll()* functions are added for alignment with the ISO/IEC 9899:1999 standard.5986 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
5987 revised to align with the ISO/IEC 9899:1999 standard.5988 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
5989 marked.

5990 **NAME**

5991 cexp, cexpf, cexpl — complex exponential functions

5992 **SYNOPSIS**

5993 #include <complex.h>

5994 double complex cexp(double complex z);

5995 float complex cexpf(float complex z);

5996 long double complex cexpl(long double complex z);

5997 **DESCRIPTION**

5998 cx The functionality described on this reference page is aligned with the ISO C standard. Any
5999 conflict between the requirements described here and the ISO C standard is unintentional. This
6000 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

6001 These functions shall compute the complex exponent of z , defined as e^z .6002 **RETURN VALUE**6003 These functions shall return the complex exponential value of z .6004 **ERRORS**

6005 No errors are defined.

6006 **EXAMPLES**

6007 None.

6008 **APPLICATION USAGE**

6009 None.

6010 **RATIONALE**

6011 None.

6012 **FUTURE DIRECTIONS**

6013 None.

6014 **SEE ALSO**6015 *clog()*, the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>6016 **CHANGE HISTORY**

6017 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

6018 **NAME**

6019 cfgetispeed — get input baud rate

6020 **SYNOPSIS**

6021 #include <termios.h>

6022 speed_t cfgetispeed(const struct termios *termios_p);

6023 **DESCRIPTION**6024 The *cfgetispeed()* function shall extract the input baud rate from the **termios** structure to which
6025 the *termios_p* argument points.6026 This function shall return exactly the value in the **termios** data structure, without interpretation.6027 **RETURN VALUE**6028 Upon successful completion, *cfgetispeed()* shall return a value of type **speed_t** representing the
6029 input baud rate.6030 **ERRORS**

6031 No errors are defined.

6032 **EXAMPLES**

6033 None.

6034 **APPLICATION USAGE**

6035 None.

6036 **RATIONALE**6037 The term “baud” is used historically here, but is not technically correct. This is properly “bits
6038 per second”, which may not be the same as baud. However, the term is used because of the
6039 historical usage and understanding.6040 The *cfgetospeed()*, *cfgetispeed()*, *cfsetospeed()*, and *cfsetispeed()* functions do not take arguments as
6041 numbers, but rather as symbolic names. There are two reasons for this:

- 6042 1. Historically, numbers were not used because of the way the rate was stored in the data
-
- 6043 structure. This is retained even though a function is now used.
-
- 6044 2. More importantly, only a limited set of possible rates is at all portable, and this constrains
-
- 6045 the application to that set.

6046 There is nothing to prevent an implementation accepting as an extension a number (such as 126),
6047 and since the encoding of the Bxxx symbols is not specified, this can be done to avoid
6048 introducing ambiguity.6049 Setting the input baud rate to zero was a mechanism to allow for split baud rates. Clarifications
6050 in this volume of IEEE Std 1003.1-2001 have made it possible to determine whether split rates are
6051 supported and to support them without having to treat zero as a special case. Since this
6052 functionality is also confusing, it has been declared obsolescent. The 0 argument referred to is
6053 the literal constant 0, not the symbolic constant B0. This volume of IEEE Std 1003.1-2001 does not
6054 preclude B0 from being defined as the value 0; in fact, implementations would likely benefit
6055 from the two being equivalent. This volume of IEEE Std 1003.1-2001 does not fully specify
6056 whether the previous *cfsetispeed()* value is retained after a *tcgetattr()* as the actual value or as
6057 zero. Therefore, conforming applications should always set both the input speed and output
6058 speed when setting either.6059 In historical implementations, the baud rate information is traditionally kept in **c_cflag**.
6060 Applications should be written to presume that this might be the case (and thus not blindly copy
6061 **c_cflag**), but not to rely on it in case it is in some other field of the structure. Setting the **c_cflag**
6062 field absolutely after setting a baud rate is a non-portable action because of this. In general, the

6063 unused parts of the flag fields might be used by the implementation and should not be blindly
6064 copied from the descriptions of one terminal device to another.

6065 **FUTURE DIRECTIONS**

6066 None.

6067 **SEE ALSO**

6068 *cfgetospeed()*, *cfsetispeed()*, *cfsetospeed()*, *tcgetattr()*, the Base Definitions volume of
6069 IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface, <termios.h>

6070 **CHANGE HISTORY**

6071 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

6072 **NAME**

6073 cfgetospeed — get output baud rate

6074 **SYNOPSIS**

6075 #include <termios.h>

6076 speed_t cfgetospeed(const struct termios *termios_p);

6077 **DESCRIPTION**6078 The *cfgetospeed()* function shall extract the output baud rate from the **termios** structure to which
6079 the *termios_p* argument points.6080 This function shall return exactly the value in the **termios** data structure, without interpretation.6081 **RETURN VALUE**6082 Upon successful completion, *cfgetospeed()* shall return a value of type **speed_t** representing the
6083 output baud rate.6084 **ERRORS**

6085 No errors are defined.

6086 **EXAMPLES**

6087 None.

6088 **APPLICATION USAGE**

6089 None.

6090 **RATIONALE**6091 Refer to *cfgetispeed()*.6092 **FUTURE DIRECTIONS**

6093 None.

6094 **SEE ALSO**6095 *cfgetispeed()*, *cfsetispeed()*, *cfsetospeed()*, *tcgetattr()*, the Base Definitions volume of
6096 IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface, <**termios.h**>6097 **CHANGE HISTORY**

6098 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

6099 **NAME**6100 `cfsetispeed` — set input baud rate6101 **SYNOPSIS**6102 `#include <termios.h>`6103 `int cfsetispeed(struct termios *termios_p, speed_t speed);`6104 **DESCRIPTION**6105 The `cfsetispeed()` function shall set the input baud rate stored in the structure pointed to by
6106 `termios_p` to `speed`.6107 There shall be no effect on the baud rates set in the hardware until a subsequent successful call
6108 to `tcsetattr()` with the same **termios** structure. Similarly, errors resulting from attempts to set
6109 baud rates not supported by the terminal device need not be detected until the `tcsetattr()`
6110 function is called.6111 **RETURN VALUE**6112 Upon successful completion, `cfsetispeed()` shall return 0; otherwise, -1 shall be returned, and
6113 `errno` may be set to indicate the error.6114 **ERRORS**6115 The `cfsetispeed()` function may fail if:6116 [EINVAL] The `speed` value is not a valid baud rate.6117 [EINVAL] The value of `speed` is outside the range of possible speed values as specified in
6118 `<termios.h>`.6119 **EXAMPLES**

6120 None.

6121 **APPLICATION USAGE**

6122 None.

6123 **RATIONALE**6124 Refer to `cfgetispeed()`.6125 **FUTURE DIRECTIONS**

6126 None.

6127 **SEE ALSO**6128 `cfgetispeed()`, `cfgetospeed()`, `cfsetospeed()`, `tcsetattr()`, the Base Definitions volume of
6129 IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface, `<termios.h>`6130 **CHANGE HISTORY**

6131 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

6132 **Issue 6**6133 The following new requirements on POSIX implementations derive from alignment with the
6134 Single UNIX Specification:

- 6135
- The optional setting of `errno` and the [EINVAL] error conditions are added.

6136 **NAME**

6137 cfsetospeed — set output baud rate

6138 **SYNOPSIS**

6139 #include <termios.h>

6140 int cfsetospeed(struct termios *termios_p, speed_t speed);

6141 **DESCRIPTION**

6142 The *cfsetospeed()* function shall set the output baud rate stored in the structure pointed to by
 6143 *termios_p* to *speed*.

6144 There shall be no effect on the baud rates set in the hardware until a subsequent successful call
 6145 to *tcsetattr()* with the same **termios** structure. Similarly, errors resulting from attempts to set
 6146 baud rates not supported by the terminal device need not be detected until the *tcsetattr()*
 6147 function is called.

6148 **RETURN VALUE**

6149 Upon successful completion, *cfsetospeed()* shall return 0; otherwise, it shall return -1 and *errno*
 6150 may be set to indicate the error.

6151 **ERRORS**6152 The *cfsetospeed()* function may fail if:6153 [EINVAL] The *speed* value is not a valid baud rate.

6154 [EINVAL] The value of *speed* is outside the range of possible speed values as specified in
 6155 <**termios.h**>.

6156 **EXAMPLES**

6157 None.

6158 **APPLICATION USAGE**

6159 None.

6160 **RATIONALE**6161 Refer to *cfgetispeed()*.6162 **FUTURE DIRECTIONS**

6163 None.

6164 **SEE ALSO**

6165 *cfgetispeed()*, *cfgetospeed()*, *cfsetispeed()*, *tcsetattr()*, the Base Definitions volume of
 6166 IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface, <**termios.h**>

6167 **CHANGE HISTORY**

6168 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

6169 **Issue 6**

6170 The following new requirements on POSIX implementations derive from alignment with the
 6171 Single UNIX Specification:

- 6172 • The optional setting of *errno* and the [EINVAL] error conditions are added.

6173 **NAME**

6174 chdir — change working directory

6175 **SYNOPSIS**

6176 #include <unistd.h>

6177 int chdir(const char *path);

6178 **DESCRIPTION**

6179 The *chdir()* function shall cause the directory named by the pathname pointed to by the *path*
 6180 argument to become the current working directory; that is, the starting point for path searches
 6181 for pathnames not beginning with *'/'*.

6182 **RETURN VALUE**

6183 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned, the current
 6184 working directory shall remain unchanged, and *errno* shall be set to indicate the error.

6185 **ERRORS**6186 The *chdir()* function shall fail if:

6187 [EACCES] Search permission is denied for any component of the pathname.

6188 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 6189 argument.

6190 [ENAMETOOLONG]

6191 The length of the *path* argument exceeds {PATH_MAX} or a pathname
 6192 component is longer than {NAME_MAX}.

6193 [ENOENT] A component of *path* does not name an existing directory or *path* is an empty
 6194 string.

6195 [ENOTDIR] A component of the pathname is not a directory.

6196 The *chdir()* function may fail if:

6197 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 6198 resolution of the *path* argument.

6199 [ENAMETOOLONG]

6200 As a result of encountering a symbolic link in resolution of the *path* argument,
 6201 the length of the substituted pathname string exceeded {PATH_MAX}.

6202 **EXAMPLES**6203 **Changing the Current Working Directory**

6204 The following example makes the value pointed to by **directory**, **/tmp**, the current working
 6205 directory.

6206 #include <unistd.h>

6207 ...

6208 char *directory = "/tmp";

6209 int ret;

6210 ret = chdir (directory);

6211 APPLICATION USAGE

6212 None.

6213 RATIONALE

6214 The *chdir()* function only affects the working directory of the current process.

6215 FUTURE DIRECTIONS

6216 None.

6217 SEE ALSO

6218 *getcwd()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**unistd.h**>

6219 CHANGE HISTORY

6220 First released in Issue 1. Derived from Issue 1 of the SVID.

6221 Issue 6

6222 The APPLICATION USAGE section is added.

6223 The following new requirements on POSIX implementations derive from alignment with the
6224 Single UNIX Specification:

- 6225 • The [ELOOP] mandatory error condition is added.
- 6226 • A second [ENAMETOOLONG] is added as an optional error condition.

6227 The following changes were made to align with the IEEE P1003.1a draft standard:

- 6228 • The [ELOOP] optional error condition is added.

6229 **NAME**

6230 chmod — change mode of a file

6231 **SYNOPSIS**

6232 #include <sys/stat.h>

6233 int chmod(const char *path, mode_t mode);

6234 **DESCRIPTION**

6235 XSI The *chmod()* function shall change S_ISUID, S_ISGID, S_ISVTX, and the file permission bits of
 6236 the file named by the pathname pointed to by the *path* argument to the corresponding bits in the
 6237 *mode* argument. The application shall ensure that the effective user ID of the process matches the
 6238 owner of the file or the process has appropriate privileges in order to do this.

6239 XSI S_ISUID, S_ISGID, S_ISVTX, and the file permission bits are described in <sys/stat.h>.

6240 If the calling process does not have appropriate privileges, and if the group ID of the file does
 6241 not match the effective group ID or one of the supplementary group IDs and if the file is a
 6242 regular file, bit S_ISGID (set-group-ID on execution) in the file's mode shall be cleared upon
 6243 successful return from *chmod()*.

6244 Additional implementation-defined restrictions may cause the S_ISUID and S_ISGID bits in
 6245 *mode* to be ignored.

6246 The effect on file descriptors for files open at the time of a call to *chmod()* is implementation-
 6247 defined.

6248 Upon successful completion, *chmod()* shall mark for update the *st_ctime* field of the file.

6249 **RETURN VALUE**

6250 Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned and *errno* set to
 6251 indicate the error. If -1 is returned, no change to the file mode occurs.

6252 **ERRORS**

6253 The *chmod()* function shall fail if:

6254 [EACCES] Search permission is denied on a component of the path prefix.

6255 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 6256 argument.

6257 [ENAMETOOLONG]

6258 The length of the *path* argument exceeds {PATH_MAX} or a pathname
 6259 component is longer than {NAME_MAX}.

6260 [ENOTDIR] A component of the path prefix is not a directory.

6261 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

6262 [EPERM] The effective user ID does not match the owner of the file and the process
 6263 does not have appropriate privileges.

6264 [EROFS] The named file resides on a read-only file system.

6265 The *chmod()* function may fail if:

6266 [EINTR] A signal was caught during execution of the function.

6267 [EINVAL] The value of the *mode* argument is invalid.

6268 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 6269 resolution of the *path* argument.

6270 [ENAMETOOLONG]

6271 As a result of encountering a symbolic link in resolution of the *path* argument,
6272 the length of the substituted pathname strings exceeded {PATH_MAX}.

6273 EXAMPLES

6274 Setting Read Permissions for User, Group, and Others

6275 The following example sets read permissions for the owner, group, and others.

```
6276 #include <sys/stat.h>
6277 const char *path;
6278 ...
6279 chmod(path, S_IRUSR|S_IRGRP|S_IROTH);
```

6280 Setting Read, Write, and Execute Permissions for the Owner Only

6281 The following example sets read, write, and execute permissions for the owner, and no
6282 permissions for group and others.

```
6283 #include <sys/stat.h>
6284 const char *path;
6285 ...
6286 chmod(path, S_IRWXU);
```

6287 Setting Different Permissions for Owner, Group, and Other

6288 The following example sets owner permissions for `CHANGEFILE` to read, write, and execute,
6289 group permissions to read and execute, and other permissions to read.

```
6290 #include <sys/stat.h>
6291 #define CHANGEFILE "/etc/myfile"
6292 ...
6293 chmod(CHANGEFILE, S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH);
```

6294 Setting and Checking File Permissions

6295 The following example sets the file permission bits for a file named `/home/cnd/mod1`, then calls
6296 the `stat()` function to verify the permissions.

```
6297 #include <sys/types.h>
6298 #include <sys/stat.h>
6299 int status;
6300 struct stat buffer
6301 ...
6302 chmod("home/cnd/mod1", S_IRWXU|S_IRWXG|S_IROTH|S_IWOTH);
6303 status = stat("home/cnd/mod1", &buffer);
```

6304 APPLICATION USAGE

6305 In order to ensure that the `S_ISUID` and `S_ISGID` bits are set, an application requiring this should
6306 use `stat()` after a successful `chmod()` to verify this.

6307 Any file descriptors currently open by any process on the file could possibly become invalid if
6308 the mode of the file is changed to a value which would deny access to that process. One

6309 situation where this could occur is on a stateless file system. This behavior will not occur in a
6310 conforming environment.

6311 RATIONALE

6312 This volume of IEEE Std 1003.1-2001 specifies that the S_ISGID bit is cleared by *chmod()* on a
6313 regular file under certain conditions. This is specified on the assumption that regular files may
6314 be executed, and the system should prevent users from making executable *setgid()* files perform
6315 with privileges that the caller does not have. On implementations that support execution of
6316 other file types, the S_ISGID bit should be cleared for those file types under the same
6317 circumstances.

6318 Implementations that use the S_ISUID bit to indicate some other function (for example,
6319 mandatory record locking) on non-executable files need not clear this bit on writing. They
6320 should clear the bit for executable files and any other cases where the bit grants special powers
6321 to processes that change the file contents. Similar comments apply to the S_ISGID bit.

6322 FUTURE DIRECTIONS

6323 None.

6324 SEE ALSO

6325 *chown()*, *mkdir()*, *mkfifo()*, *open()*, *stat()*, *statvfs()*, the Base Definitions volume of
6326 IEEE Std 1003.1-2001, <sys/stat.h>, <sys/types.h>

6327 CHANGE HISTORY

6328 First released in Issue 1. Derived from Issue 1 of the SVID.

6329 Issue 6

6330 The following new requirements on POSIX implementations derive from alignment with the
6331 Single UNIX Specification:

- 6332 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
6333 required for conforming implementations of previous POSIX specifications, it was not
6334 required for UNIX applications.
- 6335 • The [EINVAL] and [EINTR] optional error conditions are added.
- 6336 • A second [ENAMETOOLONG] is added as an optional error condition.

6337 The following changes were made to align with the IEEE P1003.1a draft standard:

- 6338 • The [ELOOP] optional error condition is added.

6339 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

6340 **NAME**

6341 chown — change owner and group of a file

6342 **SYNOPSIS**

6343 #include <unistd.h>

6344 int chown(const char *path, uid_t owner, gid_t group);

6345 **DESCRIPTION**6346 The *chown()* function shall change the user and group ownership of a file.6347 The *path* argument points to a pathname naming a file. The user ID and group ID of the named
6348 file shall be set to the numeric values contained in *owner* and *group*, respectively.6349 Only processes with an effective user ID equal to the user ID of the file or with appropriate
6350 privileges may change the ownership of a file. If `_POSIX_CHOWN_RESTRICTED` is in effect for
6351 *path*:

- 6352
- Changing the user ID is restricted to processes with appropriate privileges.
 - Changing the group ID is permitted to a process with an effective user ID equal to the user
6353 ID of the file, but without appropriate privileges, if and only if *owner* is equal to the file's user
6354 ID or `(uid_t)−1` and *group* is equal either to the calling process' effective group ID or to one of
6355 its supplementary group IDs.
- 6356

6357 If the specified file is a regular file, one or more of the `S_IXUSR`, `S_IXGRP`, or `S_IXOTH` bits of
6358 the file mode are set, and the process does not have appropriate privileges, the set-user-ID
6359 (`S_ISUID`) and set-group-ID (`S_ISGID`) bits of the file mode shall be cleared upon successful
6360 return from *chown()*. If the specified file is a regular file, one or more of the `S_IXUSR`, `S_IXGRP`,
6361 or `S_IXOTH` bits of the file mode are set, and the process has appropriate privileges, it is
6362 implementation-defined whether the set-user-ID and set-group-ID bits are altered. If the *chown()*
6363 function is successfully invoked on a file that is not a regular file and one or more of the
6364 `S_IXUSR`, `S_IXGRP`, or `S_IXOTH` bits of the file mode are set, the set-user-ID and set-group-ID
6365 bits may be cleared.6366 If *owner* or *group* is specified as `(uid_t)−1` or `(gid_t)−1`, respectively, the corresponding ID of the
6367 file shall not be changed. If both owner and group are `−1`, the times need not be updated.6368 Upon successful completion, *chown()* shall mark for update the *st_ctime* field of the file.6369 **RETURN VALUE**6370 Upon successful completion, 0 shall be returned; otherwise, `−1` shall be returned and *errno* set to
6371 indicate the error. If `−1` is returned, no changes are made in the user ID and group ID of the file.6372 **ERRORS**6373 The *chown()* function shall fail if:

- 6374 [EACCES] Search permission is denied on a component of the path prefix.
-
- 6375 [ELOOP] A loop exists in symbolic links encountered during resolution of the
- path*
-
- 6376 argument.
-
- 6377 [ENAMETOOLONG] The length of the
- path*
- argument exceeds
- `{PATH_MAX}`
- or a pathname
-
- 6378 component is longer than
- `{NAME_MAX}`
- .
-
- 6379 [ENOTDIR] A component of the path prefix is not a directory.
-
- 6380 [ENOENT] A component of
- path*
- does not name an existing file or
- path*
- is an empty string.
-
- 6381

6382 [EPERM] The effective user ID does not match the owner of the file, or the calling
 6383 process does not have appropriate privileges and
 6384 _POSIX_CHOWN_RESTRICTED indicates that such privilege is required.

6385 [EROFS] The named file resides on a read-only file system.

6386 The *chown()* function may fail if:

6387 [EIO] An I/O error occurred while reading or writing to the file system.

6388 [EINTR] The *chown()* function was interrupted by a signal which was caught.

6389 [EINVAL] The owner or group ID supplied is not a value supported by the
 6390 implementation.

6391 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 6392 resolution of the *path* argument.

6393 [ENAMETOOLONG]
 6394 As a result of encountering a symbolic link in resolution of the *path* argument,
 6395 the length of the substituted pathname string exceeded {PATH_MAX}.

6396 EXAMPLES

6397 None.

6398 APPLICATION USAGE

6399 Although *chown()* can be used on some implementations by the file owner to change the owner
 6400 and group to any desired values, the only portable use of this function is to change the group of
 6401 a file to the effective GID of the calling process or to a member of its group set.

6402 RATIONALE

6403 System III and System V allow a user to give away files; that is, the owner of a file may change
 6404 its user ID to anything. This is a serious problem for implementations that are intended to meet
 6405 government security regulations. Version 7 and 4.3 BSD permit only the superuser to change the
 6406 user ID of a file. Some government agencies (usually not ones concerned directly with security)
 6407 find this limitation too confining. This volume of IEEE Std 1003.1-2001 uses *may* to permit secure
 6408 implementations while not disallowing System V.

6409 System III and System V allow the owner of a file to change the group ID to anything. Version 7
 6410 permits only the superuser to change the group ID of a file. 4.3 BSD permits the owner to
 6411 change the group ID of a file to its effective group ID or to any of the groups in the list of
 6412 supplementary group IDs, but to no others.

6413 The POSIX.1-1990 standard requires that the *chown()* function invoked by a non-appropriate
 6414 privileged process clear the S_ISGID and the S_ISUID bits for regular files, and permits them to
 6415 be cleared for other types of files. This is so that changes in accessibility do not accidentally
 6416 cause files to become security holes. Unfortunately, requiring these bits to be cleared on non-
 6417 executable data files also clears the mandatory file locking bit (shared with S_ISGID), which is
 6418 an extension on many implementations (it first appeared in System V). These bits should only be
 6419 required to be cleared on regular files that have one or more of their execute bits set.

6420 FUTURE DIRECTIONS

6421 None.

6422 SEE ALSO

6423 *chmod()*, *pathconf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>,
 6424 <unistd.h>

6425 **CHANGE HISTORY**

6426 First released in Issue 1. Derived from Issue 1 of the SVID.

6427 **Issue 6**

6428 The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- 6429 • The wording describing the optional dependency on `_POSIX_CHOWN_RESTRICTED` is
6430 restored.
- 6431 • The `[EPERM]` error is restored as an error dependent on `_POSIX_CHOWN_RESTRICTED`.
6432 This is since its operand is a pathname and applications should be aware that the error may
6433 not occur for that pathname if the file system does not support
6434 `_POSIX_CHOWN_RESTRICTED`.

6435 The following new requirements on POSIX implementations derive from alignment with the
6436 Single UNIX Specification:

- 6437 • The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was
6438 required for conforming implementations of previous POSIX specifications, it was not
6439 required for UNIX applications.
- 6440 • The value for *owner* of `(uid_t)-1` allows the use of `-1` by the owner of a file to change the
6441 group ID only. A corresponding change is made for group.
- 6442 • The `[ELOOP]` mandatory error condition is added.
- 6443 • The `[EIO]` and `[EINTR]` optional error conditions are added.
- 6444 • A second `[ENAMETOOLONG]` is added as an optional error condition.

6445 The following changes were made to align with the IEEE P1003.1a draft standard:

- 6446 • Clarification is added that the `S_ISUID` and `S_ISGID` bits do not need to be cleared when the
6447 process has appropriate privileges.
- 6448 • The `[ELOOP]` optional error condition is added.

6449 **NAME**

6450 cimag, cimagf, cimagl — complex imaginary functions

6451 **SYNOPSIS**

6452 #include <complex.h>

6453 double cimag(double complex z);

6454 float cimagf(float complex z);

6455 long double cimagl(long double complex z);

6456 **DESCRIPTION**

6457 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
6458 conflict between the requirements described here and the ISO C standard is unintentional. This
6459 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

6460 These functions shall compute the imaginary part of z.

6461 **RETURN VALUE**

6462 These functions shall return the imaginary part value (as a real).

6463 **ERRORS**

6464 No errors are defined.

6465 **EXAMPLES**

6466 None.

6467 **APPLICATION USAGE**

6468 For a variable z of complex type:

6469 z == creal(z) + cimag(z)*I

6470 **RATIONALE**

6471 None.

6472 **FUTURE DIRECTIONS**

6473 None.

6474 **SEE ALSO**

6475 carg(), conj(), cproj(), creal(), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

6476 **CHANGE HISTORY**

6477 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

6478 **NAME**

6479 clearerr — clear indicators on a stream

6480 **SYNOPSIS**

6481 #include <stdio.h>

6482 void clearerr(FILE **stream*);6483 **DESCRIPTION**

6484 cx The functionality described on this reference page is aligned with the ISO C standard. Any
6485 conflict between the requirements described here and the ISO C standard is unintentional. This
6486 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

6487 The *clearerr()* function shall clear the end-of-file and error indicators for the stream to which
6488 *stream* points.

6489 **RETURN VALUE**6490 The *clearerr()* function shall not return a value.6491 **ERRORS**

6492 No errors are defined.

6493 **EXAMPLES**

6494 None.

6495 **APPLICATION USAGE**

6496 None.

6497 **RATIONALE**

6498 None.

6499 **FUTURE DIRECTIONS**

6500 None.

6501 **SEE ALSO**6502 The Base Definitions volume of IEEE Std 1003.1-2001, <**stdio.h**>6503 **CHANGE HISTORY**

6504 First released in Issue 1. Derived from Issue 1 of the SVID.

6505 **NAME**

6506 clock — report CPU time used

6507 **SYNOPSIS**

6508 #include <time.h>

6509 clock_t clock(void);

6510 **DESCRIPTION**

6511 cx The functionality described on this reference page is aligned with the ISO C standard. Any
 6512 conflict between the requirements described here and the ISO C standard is unintentional. This
 6513 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

6514 The *clock()* function shall return the implementation's best approximation to the processor time
 6515 used by the process since the beginning of an implementation-defined era related only to the
 6516 process invocation.

6517 **RETURN VALUE**

6518 To determine the time in seconds, the value returned by *clock()* should be divided by the value
 6519 xsi of the macro CLOCKS_PER_SEC. CLOCKS_PER_SEC is defined to be one million in <time.h>.
 6520 If the processor time used is not available or its value cannot be represented, the function shall
 6521 return the value (clock_t)−1.

6522 **ERRORS**

6523 No errors are defined.

6524 **EXAMPLES**

6525 None.

6526 **APPLICATION USAGE**

6527 In order to measure the time spent in a program, *clock()* should be called at the start of the
 6528 program and its return value subtracted from the value returned by subsequent calls. The value
 6529 returned by *clock()* is defined for compatibility across systems that have clocks with different
 6530 resolutions. The resolution on any particular system need not be to microsecond accuracy.

6531 The value returned by *clock()* may wrap around on some implementations. For example, on a
 6532 machine with 32-bit values for **clock_t**, it wraps after 2 147 seconds or 36 minutes.

6533 **RATIONALE**

6534 None.

6535 **FUTURE DIRECTIONS**

6536 None.

6537 **SEE ALSO**

6538 *asctime()*, *ctime()*, *difftime()*, *gmtime()*, *localtime()*, *mktime()*, *strftime()*, *strptime()*, *time()*, *utime()*,
 6539 the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>

6540 **CHANGE HISTORY**

6541 First released in Issue 1. Derived from Issue 1 of the SVID.

6542 **NAME**6543 clock_getcpuclockid — access a process CPU-time clock (**ADVANCED REALTIME**)6544 **SYNOPSIS**

6545 CPT #include <time.h>

6546 int clock_getcpuclockid(pid_t pid, clockid_t *clock_id);

6547

6548 **DESCRIPTION**

6549 The *clock_getcpuclockid()* function shall return the clock ID of the CPU-time clock of the process
6550 specified by *pid*. If the process described by *pid* exists and the calling process has permission,
6551 the clock ID of this clock shall be returned in *clock_id*.

6552 If *pid* is zero, the *clock_getcpuclockid()* function shall return the clock ID of the CPU-time clock of
6553 the process making the call, in *clock_id*.

6554 The conditions under which one process has permission to obtain the CPU-time clock ID of
6555 other processes are implementation-defined.

6556 **RETURN VALUE**

6557 Upon successful completion, *clock_getcpuclockid()* shall return zero; otherwise, an error number
6558 shall be returned to indicate the error.

6559 **ERRORS**

6560 The *clock_getcpuclockid()* function shall fail if:

6561 [EPERM] The requesting process does not have permission to access the CPU-time
6562 clock for the process.

6563 The *clock_getcpuclockid()* function may fail if:

6564 [ESRCH] No process can be found corresponding to the process specified by *pid*.

6565 **EXAMPLES**

6566 None.

6567 **APPLICATION USAGE**

6568 The *clock_getcpuclockid()* function is part of the Process CPU-Time Clocks option and need not
6569 be provided on all implementations.

6570 **RATIONALE**

6571 None.

6572 **FUTURE DIRECTIONS**

6573 None.

6574 **SEE ALSO**

6575 *clock_getres()*, *timer_create()*, the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>

6576 **CHANGE HISTORY**

6577 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

6578 In the SYNOPSIS, the inclusion of <sys/types.h> is no longer required.

6579 **NAME**6580 clock_getres, clock_gettime, clock_settime — clock and timer functions (**REALTIME**)6581 **SYNOPSIS**6582 TMR `#include <time.h>`

```

6583 int clock_getres(clockid_t clock_id, struct timespec *res);
6584 int clock_gettime(clockid_t clock_id, struct timespec *tp);
6585 int clock_settime(clockid_t clock_id, const struct timespec *tp);
6586

```

6587 **DESCRIPTION**

6588 The *clock_getres()* function shall return the resolution of any clock. Clock resolutions are
 6589 implementation-defined and cannot be set by a process. If the argument *res* is not NULL, the
 6590 resolution of the specified clock shall be stored in the location pointed to by *res*. If *res* is NULL,
 6591 the clock resolution is not returned. If the *time* argument of *clock_settime()* is not a multiple of *res*,
 6592 then the value is truncated to a multiple of *res*.

6593 The *clock_gettime()* function shall return the current value *tp* for the specified clock, *clock_id*.

6594 The *clock_settime()* function shall set the specified clock, *clock_id*, to the value specified by *tp*.
 6595 Time values that are between two consecutive non-negative integer multiples of the resolution
 6596 of the specified clock shall be truncated down to the smaller multiple of the resolution.

6597 A clock may be system-wide (that is, visible to all processes) or per-process (measuring time that
 6598 is meaningful only within a process). All implementations shall support a *clock_id* of
 6599 CLOCK_REALTIME as defined in <time.h>. This clock represents the realtime clock for the
 6600 system. For this clock, the values returned by *clock_gettime()* and specified by *clock_settime()*
 6601 represent the amount of time (in seconds and nanoseconds) since the Epoch. An implementation
 6602 may also support additional clocks. The interpretation of time values for these clocks is
 6603 unspecified.

6604 If the value of the CLOCK_REALTIME clock is set via *clock_settime()*, the new value of the clock
 6605 shall be used to determine the time of expiration for absolute time services based upon the
 6606 CLOCK_REALTIME clock. This applies to the time at which armed absolute timers expire. If the
 6607 absolute time requested at the invocation of such a time service is before the new value of the
 6608 clock, the time service shall expire immediately as if the clock had reached the requested time
 6609 normally.

6610 Setting the value of the CLOCK_REALTIME clock via *clock_settime()* shall have no effect on
 6611 threads that are blocked waiting for a relative time service based upon this clock, including the
 6612 *nanosleep()* function; nor on the expiration of relative timers based upon this clock.
 6613 Consequently, these time services shall expire when the requested relative interval elapses,
 6614 independently of the new or old value of the clock.

6615 MON If the Monotonic Clock option is supported, all implementations shall support a *clock_id* of
 6616 CLOCK_MONOTONIC defined in <time.h>. This clock represents the monotonic clock for the
 6617 system. For this clock, the value returned by *clock_gettime()* represents the amount of time (in
 6618 seconds and nanoseconds) since an unspecified point in the past (for example, system start-up
 6619 time, or the Epoch). This point does not change after system start-up time. The value of the
 6620 CLOCK_MONOTONIC clock cannot be set via *clock_settime()*. This function shall fail if it is
 6621 invoked with a *clock_id* argument of CLOCK_MONOTONIC.

6622 The effect of setting a clock via *clock_settime()* on armed per-process timers associated with a
 6623 clock other than CLOCK_REALTIME is implementation-defined.

6624 CS If the value of the CLOCK_REALTIME clock is set via *clock_settime()*, the new value of the clock
 6625 shall be used to determine the time at which the system shall awaken a thread blocked on an

absolute *clock_nanosleep()* call based upon the `CLOCK_REALTIME` clock. If the absolute time requested at the invocation of such a time service is before the new value of the clock, the call shall return immediately as if the clock had reached the requested time normally.

Setting the value of the `CLOCK_REALTIME` clock via *clock_settime()* shall have no effect on any thread that is blocked on a relative *clock_nanosleep()* call. Consequently, the call shall return when the requested relative interval elapses, independently of the new or old value of the clock.

The appropriate privilege to set a particular clock is implementation-defined.

CPT If `_POSIX_CPUTIME` is defined, implementations shall support clock ID values obtained by invoking *clock_getcpuclockid()*, which represent the CPU-time clock of a given process. Implementations shall also support the special `clockid_t` value `CLOCK_PROCESS_CPUTIME_ID`, which represents the CPU-time clock of the calling process when invoking one of the *clock_**() or *timer_**() functions. For these clock IDs, the values returned by *clock_gettime()* and specified by *clock_settime()* represent the amount of execution time of the process associated with the clock. Changing the value of a CPU-time clock via *clock_settime()* shall have no effect on the behavior of the sporadic server scheduling policy (see **Scheduling Policies** (on page 44)).

TCT If `_POSIX_THREAD_CPUTIME` is defined, implementations shall support clock ID values obtained by invoking *pthread_getcpuclockid()*, which represent the CPU-time clock of a given thread. Implementations shall also support the special `clockid_t` value `CLOCK_THREAD_CPUTIME_ID`, which represents the CPU-time clock of the calling thread when invoking one of the *clock_**() or *timer_**() functions. For these clock IDs, the values returned by *clock_gettime()* and specified by *clock_settime()* shall represent the amount of execution time of the thread associated with the clock. Changing the value of a CPU-time clock via *clock_settime()* shall have no effect on the behavior of the sporadic server scheduling policy (see **Scheduling Policies** (on page 44)).

6651 RETURN VALUE

A return value of 0 shall indicate that the call succeeded. A return value of -1 shall indicate that an error occurred, and *errno* shall be set to indicate the error.

6654 ERRORS

The *clock_getres()*, *clock_gettime()*, and *clock_settime()* functions shall fail if:

[EINVAL] The *clock_id* argument does not specify a known clock.

The *clock_settime()* function shall fail if:

[EINVAL] The *tp* argument to *clock_settime()* is outside the range for the given clock ID.

[EINVAL] The *tp* argument specified a nanosecond value less than zero or greater than or equal to 1 000 million.

MON [EINVAL] The value of the *clock_id* argument is `CLOCK_MONOTONIC`.

The *clock_settime()* function may fail if:

[EPERM] The requesting process does not have the appropriate privilege to set the specified clock.

6665 EXAMPLES

6666 None.

6667 APPLICATION USAGE

6668 These functions are part of the Timers option and need not be available on all implementations.

6669 Note that the absolute value of the monotonic clock is meaningless (because its origin is
 6670 arbitrary), and thus there is no need to set it. Furthermore, realtime applications can rely on the
 6671 fact that the value of this clock is never set and, therefore, that time intervals measured with this
 6672 clock will not be affected by calls to *clock_settime()*.

6673 RATIONALE

6674 None.

6675 FUTURE DIRECTIONS

6676 None.

6677 SEE ALSO

6678 *clock_getcpuclockid()*, *clock_nanosleep()*, *ctime()*, *mq_timedreceive()*, *mq_timedsend()*, *nanosleep()*,
 6679 *pthread_mutex_timedlock()*, *sem_timedwait()*, *time()*, *timer_create()*, *timer_getoverrun()*, the Base
 6680 Definitions volume of IEEE Std 1003.1-2001, <time.h>

6681 CHANGE HISTORY

6682 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

6683 Issue 6

6684 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 6685 implementation does not support the Timers option.

6686 The APPLICATION USAGE section is added.

6687 The following changes were made to align with the IEEE P1003.1a draft standard:

- 6688 • Clarification is added of the effect of resetting the clock resolution.

6689 CPU-time clocks and the *clock_getcpuclockid()* function are added for alignment with
 6690 IEEE Std 1003.1d-1999.

6691 The following changes are added for alignment with IEEE Std 1003.1j-2000:

- 6692 • The DESCRIPTION is updated as follows:
 - 6693 — The value returned by *clock_gettime()* for CLOCK_MONOTONIC is specified.
 - 6694 — The *clock_settime()* function failing for CLOCK_MONOTONIC is specified.
 - 6695 — The effects of *clock_settime()* on the *clock_nanosleep()* function with respect to
 6696 CLOCK_REALTIME is specified.
- 6697 • An [EINVAL] error is added to the ERRORS section, indicating that *clock_settime()* fails for
 6698 CLOCK_MONOTONIC.
- 6699 • The APPLICATION USAGE section notes that the CLOCK_MONOTONIC clock need not
 6700 and shall not be set by *clock_settime()* since the absolute value of the CLOCK_MONOTONIC
 6701 clock is meaningless.
- 6702 • The *clock_nanosleep()*, *mq_timedreceive()*, *mq_timedsend()*, *pthread_mutex_timedlock()*,
 6703 *sem_timedwait()*, *timer_create()*, and *timer_settime()* functions are added to the SEE ALSO
 6704 section.

6705 NAME

6706 clock_nanosleep — high resolution sleep with specifiable clock (**ADVANCED REALTIME**)

6707 SYNOPSIS

6708 cs #include <time.h>

```
6709 int clock_nanosleep(clockid_t clock_id, int flags,
6710     const struct timespec *rqtp, struct timespec *rmtp);
6711
```

6712 DESCRIPTION

6713 If the flag `TIMER_ABSTIME` is not set in the *flags* argument, the *clock_nanosleep()* function shall
 6714 cause the current thread to be suspended from execution until either the time interval specified
 6715 by the *rqtp* argument has elapsed, or a signal is delivered to the calling thread and its action is to
 6716 invoke a signal-catching function, or the process is terminated. The clock used to measure the
 6717 time shall be the clock specified by *clock_id*.

6718 If the flag `TIMER_ABSTIME` is set in the *flags* argument, the *clock_nanosleep()* function shall
 6719 cause the current thread to be suspended from execution until either the time value of the clock
 6720 specified by *clock_id* reaches the absolute time specified by the *rqtp* argument, or a signal is
 6721 delivered to the calling thread and its action is to invoke a signal-catching function, or the
 6722 process is terminated. If, at the time of the call, the time value specified by *rqtp* is less than or
 6723 equal to the time value of the specified clock, then *clock_nanosleep()* shall return immediately
 6724 and the calling process shall not be suspended.

6725 The suspension time caused by this function may be longer than requested because the
 6726 argument value is rounded up to an integer multiple of the sleep resolution, or because of the
 6727 scheduling of other activity by the system. But, except for the case of being interrupted by a
 6728 signal, the suspension time for the relative *clock_nanosleep()* function (that is, with the
 6729 `TIMER_ABSTIME` flag not set) shall not be less than the time interval specified by *rqtp*, as
 6730 measured by the corresponding clock. The suspension for the absolute *clock_nanosleep()* function
 6731 (that is, with the `TIMER_ABSTIME` flag set) shall be in effect at least until the value of the
 6732 corresponding clock reaches the absolute time specified by *rqtp*, except for the case of being
 6733 interrupted by a signal.

6734 The use of the *clock_nanosleep()* function shall have no effect on the action or blockage of any
 6735 signal.

6736 The *clock_nanosleep()* function shall fail if the *clock_id* argument refers to the CPU-time clock of
 6737 the calling thread. It is unspecified whether *clock_id* values of other CPU-time clocks are
 6738 allowed.

6739 RETURN VALUE

6740 If the *clock_nanosleep()* function returns because the requested time has elapsed, its return value
 6741 shall be zero.

6742 If the *clock_nanosleep()* function returns because it has been interrupted by a signal, it shall return
 6743 the corresponding error value. For the relative *clock_nanosleep()* function, if the *rmtp* argument is
 6744 non-NULL, the **timespec** structure referenced by it shall be updated to contain the amount of
 6745 time remaining in the interval (the requested time minus the time actually slept). If the *rmtp*
 6746 argument is NULL, the remaining time is not returned. The absolute *clock_nanosleep()* function
 6747 has no effect on the structure referenced by *rmtp*.

6748 If *clock_nanosleep()* fails, it shall return the corresponding error value.

6749 **ERRORS**6750 The *clock_nanosleep()* function shall fail if:6751 [EINTR] The *clock_nanosleep()* function was interrupted by a signal.

6752 [EINVAL] The *rtp* argument specified a nanosecond value less than zero or greater than
6753 or equal to 1 000 million; or the `TIMER_ABSTIME` flag was specified in *flags*
6754 and the *rtp* argument is outside the range for the clock specified by *clock_id*;
6755 or the *clock_id* argument does not specify a known clock, or specifies the
6756 CPU-time clock of the calling thread.

6757 [ENOTSUP] The *clock_id* argument specifies a clock for which *clock_nanosleep()* is not
6758 supported, such as a CPU-time clock.

6759 **EXAMPLES**

6760 None.

6761 **APPLICATION USAGE**

6762 Calling *clock_nanosleep()* with the value `TIMER_ABSTIME` not set in the *flags* argument and with
6763 a *clock_id* of `CLOCK_REALTIME` is equivalent to calling *nanosleep()* with the same *rtp* and *rmtp*
6764 arguments.

6765 **RATIONALE**

6766 The *nanosleep()* function specifies that the system-wide clock `CLOCK_REALTIME` is used to
6767 measure the elapsed time for this time service. However, with the introduction of the monotonic
6768 clock `CLOCK_MONOTONIC` a new relative sleep function is needed to allow an application to
6769 take advantage of the special characteristics of this clock.

6770 There are many applications in which a process needs to be suspended and then activated
6771 multiple times in a periodic way; for example, to poll the status of a non-interrupting device or
6772 to refresh a display device. For these cases, it is known that precise periodic activation cannot be
6773 achieved with a relative *sleep()* or *nanosleep()* function call. Suppose, for example, a periodic
6774 process that is activated at time T_0 , executes for a while, and then wants to suspend itself until
6775 time T_0+T , the period being T . If this process wants to use the *nanosleep()* function, it must first
6776 call *clock_gettime()* to get the current time, then calculate the difference between the current time
6777 and T_0+T and, finally, call *nanosleep()* using the computed interval. However, the process could
6778 be preempted by a different process between the two function calls, and in this case the interval
6779 computed would be wrong; the process would wake up later than desired. This problem would
6780 not occur with the absolute *clock_nanosleep()* function, since only one function call would be
6781 necessary to suspend the process until the desired time. In other cases, however, a relative sleep
6782 is needed, and that is why both functionalities are required.

6783 Although it is possible to implement periodic processes using the timers interface, this
6784 implementation would require the use of signals, and the reservation of some signal numbers. In
6785 this regard, the reasons for including an absolute version of the *clock_nanosleep()* function in
6786 IEEE Std 1003.1-2001 are the same as for the inclusion of the relative *nanosleep()*.

6787 It is also possible to implement precise periodic processes using *pthread_cond_timedwait()*, in
6788 which an absolute timeout is specified that takes effect if the condition variable involved is
6789 never signaled. However, the use of this interface is unnatural, and involves performing other
6790 operations on mutexes and condition variables that imply an unnecessary overhead.
6791 Furthermore, *pthread_cond_timedwait()* is not available in implementations that do not support
6792 threads.

6793 Although the interface of the relative and absolute versions of the new high resolution sleep
6794 service is the same *clock_nanosleep()* function, the *rmtp* argument is only used in the relative
6795 sleep. This argument is needed in the relative *clock_nanosleep()* function to reissue the function

6796 call if it is interrupted by a signal, but it is not needed in the absolute *clock_nanosleep()* function
6797 call; if the call is interrupted by a signal, the absolute *clock_nanosleep()* function can be invoked
6798 again with the same *rqtp* argument used in the interrupted call.

6799 **FUTURE DIRECTIONS**

6800 None.

6801 **SEE ALSO**

6802 *clock_getres()*, *nanosleep()*, *pthread_cond_timedwait()*, *sleep()*, the Base Definitions volume of
6803 IEEE Std 1003.1-2001, <**time.h**>

6804 **CHANGE HISTORY**

6805 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

6806 **NAME**6807 clock_settime — clock and timer functions (**REALTIME**)6808 **SYNOPSIS**6809 TMR `#include <time.h>`6810 `int clock_settime(clockid_t clock_id, const struct timespec *tp);`

6811

6812 **DESCRIPTION**6813 Refer to *clock_getres()*.

6814 **NAME**

6815 clog, clogf, clogl — complex natural logarithm functions

6816 **SYNOPSIS**

6817 #include <complex.h>

6818 double complex clog(double complex z);

6819 float complex clogf(float complex z);

6820 long double complex clogl(long double complex z);

6821 **DESCRIPTION**

6822 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
6823 conflict between the requirements described here and the ISO C standard is unintentional. This
6824 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

6825 These functions shall compute the complex natural (base *e*) logarithm of *z*, with a branch cut
6826 along the negative real axis.

6827 **RETURN VALUE**

6828 These functions shall return the complex natural logarithm value, in the range of a strip
6829 mathematically unbounded along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary
6830 axis.

6831 **ERRORS**

6832 No errors are defined.

6833 **EXAMPLES**

6834 None.

6835 **APPLICATION USAGE**

6836 None.

6837 **RATIONALE**

6838 None.

6839 **FUTURE DIRECTIONS**

6840 None.

6841 **SEE ALSO**6842 *cexp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>6843 **CHANGE HISTORY**

6844 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

6845 **NAME**

6846 close — close a file descriptor

6847 **SYNOPSIS**

6848 #include <unistd.h>

6849 int close(int *fd*);6850 **DESCRIPTION**

6851 The *close()* function shall deallocate the file descriptor indicated by *fd*. To deallocate means
 6852 to make the file descriptor available for return by subsequent calls to *open()* or other functions
 6853 that allocate file descriptors. All outstanding record locks owned by the process on the file
 6854 associated with the file descriptor shall be removed (that is, unlocked).

6855 If *close()* is interrupted by a signal that is to be caught, it shall return -1 with *errno* set to [EINTR]
 6856 and the state of *fd* is unspecified. If an I/O error occurred while reading from or writing to the
 6857 file system during *close()*, it may return -1 with *errno* set to [EIO]; if this error is returned, the
 6858 state of *fd* is unspecified.

6859 When all file descriptors associated with a pipe or FIFO special file are closed, any data
 6860 remaining in the pipe or FIFO shall be discarded.

6861 When all file descriptors associated with an open file description have been closed, the open file
 6862 description shall be freed.

6863 If the link count of the file is 0, when all file descriptors associated with the file are closed, the
 6864 space occupied by the file shall be freed and the file shall no longer be accessible.

6865 XSR If a STREAMS-based *fd* is closed and the calling process was previously registered to receive
 6866 a SIGPOLL signal for events associated with that STREAM, the calling process shall be
 6867 unregistered for events associated with the STREAM. The last *close()* for a STREAM shall cause
 6868 the STREAM associated with *fd* to be dismantled. If O_NONBLOCK is not set and there have
 6869 been no signals posted for the STREAM, and if there is data on the module's write queue, *close()*
 6870 shall wait for an unspecified time (for each module and driver) for any output to drain before
 6871 dismantling the STREAM. The time delay can be changed via an I_SETCLTIME *ioctl()* request. If
 6872 the O_NONBLOCK flag is set, or if there are any pending signals, *close()* shall not wait for
 6873 output to drain, and shall dismantle the STREAM immediately.

6874 If the implementation supports STREAMS-based pipes, and *fd* is associated with one end of a
 6875 pipe, the last *close()* shall cause a hangup to occur on the other end of the pipe. In addition, if the
 6876 other end of the pipe has been named by *fattach()*, then the last *close()* shall force the named end
 6877 to be detached by *fdetach()*. If the named end has no open file descriptors associated with it and
 6878 gets detached, the STREAM associated with that end shall also be dismantled.

6879 XSI If *fd* refers to the master side of a pseudo-terminal, and this is the last close, a SIGHUP signal
 6880 shall be sent to the process group, if any, for which the slave side of the pseudo-terminal is the
 6881 controlling terminal. It is unspecified whether closing the master side of the pseudo-terminal
 6882 flushes all queued input and output.

6883 XSR If *fd* refers to the slave side of a STREAMS-based pseudo-terminal, a zero-length message
 6884 may be sent to the master.

6885 AIO When there is an outstanding cancelable asynchronous I/O operation against *fd* when *close()*
 6886 is called, that I/O operation may be canceled. An I/O operation that is not canceled completes
 6887 as if the *close()* operation had not yet occurred. All operations that are not canceled shall
 6888 complete as if the *close()* blocked until the operations completed. The *close()* operation itself
 6889 need not block awaiting such I/O completion. Whether any I/O operation is canceled, and
 6890 which I/O operation may be canceled upon *close()*, is implementation-defined.

6891 MF|SHM If a shared memory object or a memory mapped file remains referenced at the last close (that is,
 6892 a process has it mapped), then the entire contents of the memory object shall persist until the
 6893 memory object becomes unreferenced. If this is the last close of a shared memory object or a
 6894 memory mapped file and the close results in the memory object becoming unreferenced, and the
 6895 memory object has been unlinked, then the memory object shall be removed.

6896 If *fdes* refers to a socket, *close()* shall cause the socket to be destroyed. If the socket is in
 6897 connection-mode, and the SO_LINGER option is set for the socket with non-zero linger time,
 6898 and the socket has untransmitted data, then *close()* shall block for up to the current linger
 6899 interval until all data is transmitted.

6900 RETURN VALUE

6901 Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned and *errno* set to
 6902 indicate the error.

6903 ERRORS

6904 The *close()* function shall fail if:

6905 [EBADF] The *fdes* argument is not a valid file descriptor.

6906 [EINTR] The *close()* function was interrupted by a signal.

6907 The *close()* function may fail if:

6908 [EIO] An I/O error occurred while reading from or writing to the file system.

6909 EXAMPLES

6910 Reassigning a File Descriptor

6911 The following example closes the file descriptor associated with standard output for the current
 6912 process, re-assigns standard output to a new file descriptor, and closes the original file
 6913 descriptor to clean up. This example assumes that the file descriptor 0 (which is the descriptor
 6914 for standard input) is not closed.

```
6915 #include <unistd.h>
6916 ...
6917 int pfd;
6918 ...
6919 close(1);
6920 dup(pfd);
6921 close(pfd);
6922 ...
```

6923 Incidentally, this is exactly what could be achieved using:

```
6924 dup2(pfd, 1);
6925 close(pfd);
```

6926 Closing a File Descriptor

6927 In the following example, *close()* is used to close a file descriptor after an unsuccessful attempt is
 6928 made to associate that file descriptor with a stream.

```
6929 #include <stdio.h>
6930 #include <unistd.h>
6931 #include <stdlib.h>
```



```

6932     #define LOCKFILE "/etc/ptmp"
6933     ...
6934     int pfd;
6935     FILE *fpfd;
6936     ...
6937     if ((fpfd = fdopen (pfd, "w")) == NULL) {
6938         close(pfd);
6939         unlink(LOCKFILE);
6940         exit(1);
6941     }
6942     ...

```

6943 APPLICATION USAGE

6944 An application that had used the *stdio* routine *fopen()* to open a file should use the
 6945 corresponding *fclose()* routine rather than *close()*. Once a file is closed, the file descriptor no
 6946 longer exists, since the integer corresponding to it no longer refers to a file.

6947 RATIONALE

6948 The use of interruptible device close routines should be discouraged to avoid problems with the
 6949 implicit closes of file descriptors by *exec* and *exit()*. This volume of IEEE Std 1003.1-2001 only
 6950 intends to permit such behavior by specifying the [EINTR] error condition.

6951 FUTURE DIRECTIONS

6952 None.

6953 SEE ALSO

6954 Section 2.6 (on page 38), *fattach()*, *fclose()*, *fdetach()*, *fopen()*, *ioctl()*, *open()*, the Base Definitions
 6955 volume of IEEE Std 1003.1-2001, <unistd.h>

6956 CHANGE HISTORY

6957 First released in Issue 1. Derived from Issue 1 of the SVID.

6958 Issue 5

6959 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

6960 Issue 6

6961 The DESCRIPTION related to a STREAMS-based file or pseudo-terminal is marked as part of the
 6962 XSI STREAMS Option Group.

6963 The following new requirements on POSIX implementations derive from alignment with the
 6964 Single UNIX Specification:

- 6965 • The [EIO] error condition is added as an optional error.
- 6966 • The DESCRIPTION is updated to describe the state of the *fildev* file descriptor as unspecified
 6967 if an I/O error occurs and an [EIO] error condition is returned.

6968 Text referring to sockets is added to the DESCRIPTION.

6969 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that
 6970 shared memory objects and memory mapped files (and not typed memory objects) are the types
 6971 of memory objects to which the paragraph on last closes applies.

6972 **NAME**

6973 closedir — close a directory stream

6974 **SYNOPSIS**

6975 #include <dirent.h>

6976 int closedir(DIR *dirp);

6977 **DESCRIPTION**

6978 The *closedir()* function shall close the directory stream referred to by the argument *dirp*. Upon
 6979 return, the value of *dirp* may no longer point to an accessible object of the type **DIR**. If a file
 6980 descriptor is used to implement type **DIR**, that file descriptor shall be closed.

6981 **RETURN VALUE**

6982 Upon successful completion, *closedir()* shall return 0; otherwise, -1 shall be returned and *errno*
 6983 set to indicate the error.

6984 **ERRORS**6985 The *closedir()* function may fail if:6986 [EBADF] The *dirp* argument does not refer to an open directory stream.6987 [EINTR] The *closedir()* function was interrupted by a signal.6988 **EXAMPLES**6989 **Closing a Directory Stream**6990 The following program fragment demonstrates how the *closedir()* function is used.

```

6991       ...
6992       DIR *dir;
6993       struct dirent *dp;
6994       ...
6995       if ((dir = opendir(".")) == NULL) {
6996       ...
6997       }
6998       while ((dp = readdir(dir)) != NULL) {
6999       ...
7000       }
7001       closedir(dir);
7002       ...

```

7003 **APPLICATION USAGE**

7004 None.

7005 **RATIONALE**

7006 None.

7007 **FUTURE DIRECTIONS**

7008 None.

7009 **SEE ALSO**7010 *opendir()*, the Base Definitions volume of IEEE Std 1003.1-2001, <dirent.h>

7011 **CHANGE HISTORY**

7012 First released in Issue 2.

7013 **Issue 6**7014 In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.7015 The following new requirements on POSIX implementations derive from alignment with the
7016 Single UNIX Specification:

- 7017 • The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was
7018 required for conforming implementations of previous POSIX specifications, it was not
7019 required for UNIX applications.
- 7020 • The [EINTR] error condition is added as an optional error condition.

7021 NAME

7022 closelog, openlog, setlogmask, syslog — control system log

7023 SYNOPSIS

```
7024 xSI    #include <syslog.h>

7025        void closelog(void);
7026        void openlog(const char *ident, int logopt, int facility);
7027        int setlogmask(int maskpri);
7028        void syslog(int priority, const char *message, ... /* arguments */);
7029
```

7030 DESCRIPTION

7031 The *syslog()* function shall send a message to an implementation-defined logging facility, which
 7032 may log it in an implementation-defined system log, write it to the system console, forward it to
 7033 a list of users, or forward it to the logging facility on another host over the network. The logged
 7034 message shall include a message header and a message body. The message header contains at
 7035 least a timestamp and a tag string.

7036 The message body is generated from the *message* and following arguments in the same manner
 7037 as if these were arguments to *printf()*, except that the additional conversion specification *%m*
 7038 shall be recognized; it shall convert no arguments, shall cause the output of the error message
 7039 string associated with the value of *errno* on entry to *syslog()*, and may be mixed with argument
 7040 specifications of the "*%n\$*" form. If a complete conversion specification with the *m* conversion
 7041 specifier character is not just *%m*, the behavior is undefined. A trailing <newline> may be added
 7042 if needed.

7043 Values of the *priority* argument are formed by OR'ing together a severity-level value and an
 7044 optional facility value. If no facility value is specified, the current default facility value is used.

7045 Possible values of severity level include:

7046	LOG_EMERG	A panic condition.
7047	LOG_ALERT	A condition that should be corrected immediately, such as a corrupted system
7048		database.
7049	LOG_CRIT	Critical conditions, such as hard device errors.
7050	LOG_ERR	Errors.
7051	LOG_WARNING	
7052		Warning messages.
7053	LOG_NOTICE	Conditions that are not error conditions, but that may require special
7054		handling.
7055	LOG_INFO	Informational messages.
7056	LOG_DEBUG	Messages that contain information normally of use only when debugging a
7057		program.

7058 The facility indicates the application or system component generating the message. Possible
 7059 facility values include:

7060	LOG_USER	Messages generated by arbitrary processes. This is the default facility
7061		identifier if none is specified.
7062	LOG_LOCAL0	Reserved for local use.

7063	LOG_LOCAL1	Reserved for local use.
7064	LOG_LOCAL2	Reserved for local use.
7065	LOG_LOCAL3	Reserved for local use.
7066	LOG_LOCAL4	Reserved for local use.
7067	LOG_LOCAL5	Reserved for local use.
7068	LOG_LOCAL6	Reserved for local use.
7069	LOG_LOCAL7	Reserved for local use.
7070	The <i>openlog()</i> function shall set process attributes that affect subsequent calls to <i>syslog()</i> . The	
7071	<i>ident</i> argument is a string that is prepended to every message. The <i>logopt</i> argument indicates	
7072	logging options. Values for <i>logopt</i> are constructed by a bitwise-inclusive OR of zero or more of	
7073	the following:	
7074	LOG_PID	Log the process ID with each message. This is useful for identifying specific
7075		processes.
7076	LOG_CONS	Write messages to the system console if they cannot be sent to the logging
7077		facility. The <i>syslog()</i> function ensures that the process does not acquire the
7078		console as a controlling terminal in the process of writing the message.
7079	LOG_NDELAY	Open the connection to the logging facility immediately. Normally the open is
7080		delayed until the first message is logged. This is useful for programs that need
7081		to manage the order in which file descriptors are allocated.
7082	LOG_ODELAY	Delay open until <i>syslog()</i> is called.
7083	LOG_NOWAIT	Do not wait for child processes that may have been created during the course
7084		of logging the message. This option should be used by processes that enable
7085		notification of child termination using SIGCHLD, since <i>syslog()</i> may
7086		otherwise block waiting for a child whose exit status has already been
7087		collected.
7088	The <i>facility</i> argument encodes a default facility to be assigned to all messages that do not have	
7089	an explicit facility already encoded. The initial default facility is LOG_USER.	
7090	The <i>openlog()</i> and <i>syslog()</i> functions may allocate a file descriptor. It is not necessary to call	
7091	<i>openlog()</i> prior to calling <i>syslog()</i> .	
7092	The <i>closelog()</i> function shall close any open file descriptors allocated by previous calls to	
7093	<i>openlog()</i> or <i>syslog()</i> .	
7094	The <i>setlogmask()</i> function shall set the log priority mask for the current process to <i>maskpri</i> and	
7095	return the previous mask. If the <i>maskpri</i> argument is 0, the current log mask is not modified.	
7096	Calls by the current process to <i>syslog()</i> with a priority not set in <i>maskpri</i> shall be rejected. The	
7097	default log mask allows all priorities to be logged. A call to <i>openlog()</i> is not required prior to	
7098	calling <i>setlogmask()</i> .	
7099	Symbolic constants for use as values of the <i>logopt</i> , <i>facility</i> , <i>priority</i> , and <i>maskpri</i> arguments are	
7100	defined in the <syslog.h> header.	
7101	RETURN VALUE	
7102	The <i>setlogmask()</i> function shall return the previous log priority mask. The <i>closelog()</i> , <i>openlog()</i> ,	
7103	and <i>syslog()</i> functions shall not return a value.	

7104 **ERRORS**

7105 No errors are defined.

7106 **EXAMPLES**7107 **Using openlog()**

7108 The following example causes subsequent calls to *syslog()* to log the process ID with each message, and to write messages to the system console if they cannot be sent to the logging facility.

```
7111       #include <syslog.h>
7112
7112       char *ident = "Process demo";
7113       int logopt = LOG_PID | LOG_CONS;
7114       int facility = LOG_USER;
7115       ...
7116       openlog(ident, logopt, facility);
```

7117 **Using setlogmask()**

7118 The following example causes subsequent calls to *syslog()* to accept error messages or messages generated by arbitrary processes, and to reject all other messages.

```
7120       #include <syslog.h>
7121
7121       int result;
7122       int mask = LOG_MASK (LOG_ERR | LOG_USER);
7123       ...
7124       result = setlogmask(mask);
```

7125 **Using syslog**

7126 The following example sends the message "This is a message" to the default logging facility, marking the message as an error message generated by random processes.

```
7128       #include <syslog.h>
7129
7129       char *message = "This is a message";
7130       int priority = LOG_ERR | LOG_USER;
7131       ...
7132       syslog(priority, message);
```

7133 **APPLICATION USAGE**

7134 None.

7135 **RATIONALE**

7136 None.

7137 **FUTURE DIRECTIONS**

7138 None.

7139 **SEE ALSO**7140 *printf()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<syslog.h>**

7141 **CHANGE HISTORY**

7142 First released in Issue 4, Version 2.

7143 **Issue 5**

7144 Moved from X/OPEN UNIX extension to BASE.

7145 **NAME**

7146 confstr — get configurable variables

7147 **SYNOPSIS**

7148 #include <unistd.h>

7149 size_t confstr(int *name*, char **buf*, size_t *len*);7150 **DESCRIPTION**7151 The *confstr()* function shall return configuration-defined string values. Its use and purpose are
7152 similar to *sysconf()*, but it is used where string values rather than numeric values are returned.7153 The *name* argument represents the system variable to be queried. The implementation shall
7154 support the following name values, defined in <unistd.h>. It may support others:

7155 _CS_PATH
 7156 _CS_POSIX_V6_ILP32_OFF32_CFLAGS
 7157 _CS_POSIX_V6_ILP32_OFF32_LDFLAGS
 7158 _CS_POSIX_V6_ILP32_OFF32_LIBS
 7159 _CS_POSIX_V6_ILP32_OFFBIG_CFLAGS
 7160 _CS_POSIX_V6_ILP32_OFFBIG_LDFLAGS
 7161 _CS_POSIX_V6_ILP32_OFFBIG_LIBS
 7162 _CS_POSIX_V6_LP64_OFF64_CFLAGS
 7163 _CS_POSIX_V6_LP64_OFF64_LDFLAGS
 7164 _CS_POSIX_V6_LP64_OFF64_LIBS
 7165 _CS_POSIX_V6_LPBIG_OFFBIG_CFLAGS
 7166 _CS_POSIX_V6_LPBIG_OFFBIG_LDFLAGS
 7167 _CS_POSIX_V6_LPBIG_OFFBIG_LIBS
 7168 _CS_POSIX_V6_WIDTH_RESTRICTED_ENVS
 7169 XSI CS_XBS5_ILP32_OFF32_CFLAGS (LEGACY)
 7170 CS_XBS5_ILP32_OFF32_LDFLAGS (LEGACY)
 7171 CS_XBS5_ILP32_OFF32_LIBS (LEGACY)
 7172 CS_XBS5_ILP32_OFF32_LINTFLAGS (LEGACY)
 7173 CS_XBS5_ILP32_OFFBIG_CFLAGS (LEGACY)
 7174 CS_XBS5_ILP32_OFFBIG_LDFLAGS (LEGACY)
 7175 CS_XBS5_ILP32_OFFBIG_LIBS (LEGACY)
 7176 CS_XBS5_ILP32_OFFBIG_LINTFLAGS (LEGACY)
 7177 CS_XBS5_LP64_OFF64_CFLAGS (LEGACY)
 7178 CS_XBS5_LP64_OFF64_LDFLAGS (LEGACY)
 7179 CS_XBS5_LP64_OFF64_LIBS (LEGACY)
 7180 CS_XBS5_LP64_OFF64_LINTFLAGS (LEGACY)
 7181 CS_XBS5_LPBIG_OFFBIG_CFLAGS (LEGACY)
 7182 CS_XBS5_LPBIG_OFFBIG_LDFLAGS (LEGACY)
 7183 CS_XBS5_LPBIG_OFFBIG_LIBS (LEGACY)
 7184 CS_XBS5_LPBIG_OFFBIG_LINTFLAGS (LEGACY)

7185

7186 If *len* is not 0, and if *name* has a configuration-defined value, *confstr()* shall copy that value into
 7187 the *len*-byte buffer pointed to by *buf*. If the string to be returned is longer than *len* bytes,
 7188 including the terminating null, then *confstr()* shall truncate the string to *len*–1 bytes and null-
 7189 terminate the result. The application can detect that the string was truncated by comparing the
 7190 value returned by *confstr()* with *len*.

7191 If *len* is 0 and *buf* is a null pointer, then *confstr()* shall still return the integer value as defined
 7192 below, but shall not return a string. If *len* is 0 but *buf* is not a null pointer, the result is
 7193 unspecified.

7194 If the implementation supports the POSIX shell option, the string stored in *buf* after a call to:

7195 `confstr(_CS_PATH, buf, sizeof(buf))`

7196 can be used as a value of the *PATH* environment variable that accesses all of the standard
7197 utilities of IEEE Std 1003.1-2001, if the return value is less than or equal to *sizeof(buf)*.

7198 RETURN VALUE

7199 If *name* has a configuration-defined value, *confstr()* shall return the size of buffer that would be
7200 needed to hold the entire configuration-defined value including the terminating null. If this
7201 return value is greater than *len*, the string returned in *buf* is truncated.

7202 If *name* is invalid, *confstr()* shall return 0 and set *errno* to indicate the error.

7203 If *name* does not have a configuration-defined value, *confstr()* shall return 0 and leave *errno*
7204 unchanged.

7205 ERRORS

7206 The *confstr()* function shall fail if:

7207 [EINVAL] The value of the *name* argument is invalid.

7208 EXAMPLES

7209 None.

7210 APPLICATION USAGE

7211 An application can distinguish between an invalid *name* parameter value and one that
7212 corresponds to a configurable variable that has no configuration-defined value by checking if
7213 *errno* is modified. This mirrors the behavior of *sysconf()*.

7214 The original need for this function was to provide a way of finding the configuration-defined
7215 default value for the environment variable *PATH*. Since *PATH* can be modified by the user to
7216 include directories that could contain utilities replacing the standard utilities in the Shell and
7217 Utilities volume of IEEE Std 1003.1-2001, applications need a way to determine the system-
7218 supplied *PATH* environment variable value that contains the correct search path for the standard
7219 utilities.

7220 An application could use:

7221 `confstr(name, (char *)NULL, (size_t)0)`

7222 to find out how big a buffer is needed for the string value; use *malloc()* to allocate a buffer to
7223 hold the string; and call *confstr()* again to get the string. Alternately, it could allocate a fixed,
7224 static buffer that is big enough to hold most answers (perhaps 512 or 1024 bytes), but then use
7225 *malloc()* to allocate a larger buffer if it finds that this is too small.

7226 RATIONALE

7227 Application developers can normally determine any configuration variable by means of reading
7228 from the stream opened by a call to:

7229 `popen("command -p getconf variable", "r");`

7230 The *confstr()* function with a *name* argument of *_CS_PATH* returns a string that can be used as a
7231 *PATH* environment variable setting that will reference the standard shell and utilities as
7232 described in the Shell and Utilities volume of IEEE Std 1003.1-2001.

7233 The *confstr()* function copies the returned string into a buffer supplied by the application instead
7234 of returning a pointer to a string. This allows a cleaner function in some implementations (such
7235 as those with lightweight threads) and resolves questions about when the application must copy
7236 the string returned.

7237 **FUTURE DIRECTIONS**

7238 None.

7239 **SEE ALSO**

7240 *pathconf()*, *sysconf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**unistd.h**>, the Shell
7241 and Utilities volume of IEEE Std 1003.1-2001, *c99*

7242 **CHANGE HISTORY**

7243 First released in Issue 4. Derived from the ISO POSIX-2 standard.

7244 **Issue 5**

7245 A table indicating the permissible values of *name* is added to the DESCRIPTION. All those
7246 marked EX are new in this issue.

7247 **Issue 6**

7248 The Open Group Corrigendum U033/7 is applied. The return value for the case returning the
7249 size of the buffer now explicitly states that this includes the terminating null.

7250 The following new requirements on POSIX implementations derive from alignment with the
7251 Single UNIX Specification:

- 7252 • The DESCRIPTION is updated with new arguments which can be used to determine
7253 configuration strings for C compiler flags, linker/loader flags, and libraries for each different
7254 supported programming environment. This is a change to support data size neutrality.

7255 The following changes were made to align with the IEEE P1003.1a draft standard:

- 7256 • The DESCRIPTION is updated to include text describing how `_CS_PATH` can be used to
7257 obtain a *PATH* to access the standard utilities.

7258 The macros associated with the *c89* programming models are marked LEGACY and new
7259 equivalent macros associated with *c99* are introduced.

7260 **NAME**

7261 conj, conjf, conjl — complex conjugate functions

7262 **SYNOPSIS**

7263 #include <complex.h>

7264 double complex conj(double complex z);

7265 float complex conjf(float complex z);

7266 long double complex conjl(long double complex z);

7267 **DESCRIPTION**

7268 cx The functionality described on this reference page is aligned with the ISO C standard. Any
7269 conflict between the requirements described here and the ISO C standard is unintentional. This
7270 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

7271 These functions shall compute the complex conjugate of *z*, by reversing the sign of its imaginary
7272 part.

7273 **RETURN VALUE**

7274 These functions return the complex conjugate value.

7275 **ERRORS**

7276 No errors are defined.

7277 **EXAMPLES**

7278 None.

7279 **APPLICATION USAGE**

7280 None.

7281 **RATIONALE**

7282 None.

7283 **FUTURE DIRECTIONS**

7284 None.

7285 **SEE ALSO**

7286 *carg()*, *cimag()*, *cproj()*, *creal()*, the Base Definitions volume of IEEE Std 1003.1-2001,
7287 <complex.h>

7288 **CHANGE HISTORY**

7289 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

7290 NAME

7291 connect — connect a socket

7292 SYNOPSIS

7293 #include <sys/socket.h>

```
7294 int connect(int socket, const struct sockaddr *address,
7295             socklen_t address_len);
```

7296 DESCRIPTION

7297 The *connect()* function shall attempt to make a connection on a socket. The function takes the
 7298 following arguments:

7299	<i>socket</i>	Specifies the file descriptor associated with the socket.
7300 7301	<i>address</i>	Points to a sockaddr structure containing the peer address. The length and format of the address depend on the address family of the socket.
7302 7303	<i>address_len</i>	Specifies the length of the sockaddr structure pointed to by the <i>address</i> argument.

7304 If the socket has not already been bound to a local address, *connect()* shall bind it to an address
 7305 which, unless the socket's address family is AF_UNIX, is an unused local address.

7306 If the initiating socket is not connection-mode, then *connect()* shall set the socket's peer address,
 7307 and no connection is made. For SOCK_DGRAM sockets, the peer address identifies where all
 7308 datagrams are sent on subsequent *send()* functions, and limits the remote sender for subsequent
 7309 *recv()* functions. If *address* is a null address for the protocol, the socket's peer address shall be
 7310 reset.

7311 If the initiating socket is connection-mode, then *connect()* shall attempt to establish a connection
 7312 to the address specified by the *address* argument. If the connection cannot be established
 7313 immediately and O_NONBLOCK is not set for the file descriptor for the socket, *connect()* shall
 7314 block for up to an unspecified timeout interval until the connection is established. If the timeout
 7315 interval expires before the connection is established, *connect()* shall fail and the connection
 7316 attempt shall be aborted. If *connect()* is interrupted by a signal that is caught while blocked
 7317 waiting to establish a connection, *connect()* shall fail and set *errno* to [EINTR], but the connection
 7318 request shall not be aborted, and the connection shall be established asynchronously.

7319 If the connection cannot be established immediately and O_NONBLOCK is set for the file
 7320 descriptor for the socket, *connect()* shall fail and set *errno* to [EINPROGRESS], but the connection
 7321 request shall not be aborted, and the connection shall be established asynchronously.
 7322 Subsequent calls to *connect()* for the same socket, before the connection is established, shall fail
 7323 and set *errno* to [EALREADY].

7324 When the connection has been established asynchronously, *select()* and *poll()* shall indicate that
 7325 the file descriptor for the socket is ready for writing.

7326 The socket in use may require the process to have appropriate privileges to use the *connect()*
 7327 function.

7328 RETURN VALUE

7329 Upon successful completion, *connect()* shall return 0; otherwise, -1 shall be returned and *errno*
 7330 set to indicate the error.

7331 ERRORS

7332 The *connect()* function shall fail if:

7333 [EADDRNOTAVAIL]

7334 The specified address is not available from the local machine.

7335	[EAFNOSUPPORT]	
7336		The specified address is not a valid address for the address family of the
7337		specified socket.
7338	[EALREADY]	A connection request is already in progress for the specified socket.
7339	[EBADF]	The <i>socket</i> argument is not a valid file descriptor.
7340	[ECONNREFUSED]	
7341		The target address was not listening for connections or refused the connection
7342		request.
7343	[EINPROGRESS]	O_NONBLOCK is set for the file descriptor for the socket and the connection
7344		cannot be immediately established; the connection shall be established
7345		asynchronously.
7346	[EINTR]	The attempt to establish a connection was interrupted by delivery of a signal
7347		that was caught; the connection shall be established asynchronously.
7348	[EISCONN]	The specified socket is connection-mode and is already connected.
7349	[ENETUNREACH]	
7350		No route to the network is present.
7351	[ENOTSOCK]	The <i>socket</i> argument does not refer to a socket.
7352	[EPROTOTYPE]	The specified address has a different type than the socket bound to the
7353		specified peer address.
7354	[ETIMEDOUT]	The attempt to connect timed out before a connection was made.
7355	If the address family of the socket is AF_UNIX, then <i>connect()</i> shall fail if:	
7356	[EIO]	An I/O error occurred while reading from or writing to the file system.
7357	[ELOOP]	A loop exists in symbolic links encountered during resolution of the pathname
7358		in <i>address</i> .
7359	[ENAMETOOLONG]	
7360		A component of a pathname exceeded {NAME_MAX} characters, or an entire
7361		pathname exceeded {PATH_MAX} characters.
7362	[ENOENT]	A component of the pathname does not name an existing file or the pathname
7363		is an empty string.
7364	[ENOTDIR]	A component of the path prefix of the pathname in <i>address</i> is not a directory.
7365	The <i>connect()</i> function may fail if:	
7366	[EACCES]	Search permission is denied for a component of the path prefix; or write
7367		access to the named socket is denied.
7368	[EADDRINUSE]	Attempt to establish a connection that uses addresses that are already in use.
7369	[ECONNRESET]	Remote host reset the connection request.
7370	[EHOSTUNREACH]	
7371		The destination host cannot be reached (probably because the host is down or
7372		a remote router cannot reach it).
7373	[EINVAL]	The <i>address_len</i> argument is not a valid length for the address family; or
7374		invalid address family in the sockaddr structure.

7375 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
7376 resolution of the pathname in *address*.

7377 [ENAMETOOLONG]
7378 Pathname resolution of a symbolic link produced an intermediate result
7379 whose length exceeds {PATH_MAX}.

7380 [ENETDOWN] The local network interface used to reach the destination is down.

7381 [ENOBUFS] No buffer space is available.

7382 [EOPNOTSUPP] The socket is listening and cannot be connected.

7383 **EXAMPLES**
7384 None.

7385 **APPLICATION USAGE**
7386 If *connect()* fails, the state of the socket is unspecified. Conforming applications should close the
7387 file descriptor and create a new socket before attempting to reconnect.

7388 **RATIONALE**
7389 None.

7390 **FUTURE DIRECTIONS**
7391 None.

7392 **SEE ALSO**
7393 *accept()*, *bind()*, *close()*, *getsockname()*, *poll()*, *select()*, *send()*, *shutdown()*, *socket()*, the Base
7394 Definitions volume of IEEE Std 1003.1-2001, <**sys/socket.h**>

7395 **CHANGE HISTORY**
7396 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
7397 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
7398 [ELOOP] error condition is added.

7399 **NAME**

7400 copysign, copysignf, copysignl — number manipulation function

7401 **SYNOPSIS**

7402 #include <math.h>

7403 double copysign(double x, double y);

7404 float copysignf(float x, float y);

7405 long double copysignl(long double x, long double y);

7406 **DESCRIPTION**

7407 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
7408 conflict between the requirements described here and the ISO C standard is unintentional. This
7409 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

7410 These functions shall produce a value with the magnitude of *x* and the sign of *y*. On
7411 implementations that represent a signed zero but do not treat negative zero consistently in
7412 arithmetic operations, these functions regard the sign of zero as positive.

7413 **RETURN VALUE**

7414 Upon successful completion, these functions shall return a value with the magnitude of *x* and
7415 the sign of *y*.

7416 **ERRORS**

7417 No errors are defined.

7418 **EXAMPLES**

7419 None.

7420 **APPLICATION USAGE**

7421 None.

7422 **RATIONALE**

7423 None.

7424 **FUTURE DIRECTIONS**

7425 None.

7426 **SEE ALSO**7427 *signbit()*, the Base Definitions volume of IEEE Std 1003.1-2001, <math.h>7428 **CHANGE HISTORY**

7429 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

7430 **NAME**

7431 cos, cosf, cosl — cosine function

7432 **SYNOPSIS**

7433 #include <math.h>

7434 double cos(double x);

7435 float cosf(float x);

7436 long double cosl(long double x);

7437 **DESCRIPTION**

7438 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 7439 conflict between the requirements described here and the ISO C standard is unintentional. This
 7440 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

7441 These functions shall compute the cosine of their argument *x*, measured in radians.

7442 An application wishing to check for error situations should set *errno* to zero and call
 7443 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 7444 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 7445 zero, an error has occurred.

7446 **RETURN VALUE**7447 Upon successful completion, these functions shall return the cosine of *x*.7448 **MX** If *x* is NaN, a NaN shall be returned.7449 If *x* is ± 0 , the value 1.0 shall be returned.

7450 If *x* is $\pm \text{Inf}$, a domain error shall occur, and either a NaN (if supported), or an implementation-
 7451 defined value shall be returned.

7452 **ERRORS**

7453 These functions shall fail if:

7454 **MX** **Domain Error** The *x* argument is $\pm \text{Inf}$.

7455 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 7456 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling
 7457 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 7458 shall be raised.

7459 **EXAMPLES**7460 **Taking the Cosine of a 45-Degree Angle**

7461 #include <math.h>

7462 ...

7463 double radians = 45 * M_PI / 180;

7464 double result;

7465 ...

7466 result = cos(radians);

7467 **APPLICATION USAGE**

7468 These functions may lose accuracy when their argument is near an odd multiple of $\pi/2$ or is far
 7469 from 0.

7470 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
 7471 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

7472 **RATIONALE**

7473 None.

7474 **FUTURE DIRECTIONS**

7475 None.

7476 **SEE ALSO**

7477 *acos()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, *sin()*, *tan()*, the Base Definitions volume of
7478 IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
7479 <math.h>

7480 **CHANGE HISTORY**

7481 First released in Issue 1. Derived from Issue 1 of the SVID.

7482 **Issue 5**

7483 The DESCRIPTION is updated to indicate how an application should check for an error. This
7484 text was previously published in the APPLICATION USAGE section.

7485 **Issue 6**7486 The *cosf()* and *cosl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

7487 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
7488 revised to align with the ISO/IEC 9899:1999 standard.

7489 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
7490 marked.

7491 **NAME**

7492 cosh, coshf, coshl — hyperbolic cosine functions

7493 **SYNOPSIS**

7494 #include <math.h>

7495 double cosh(double x);

7496 float coshf(float x);

7497 long double coshl(long double x);

7498 **DESCRIPTION**

7499 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 7500 conflict between the requirements described here and the ISO C standard is unintentional. This
 7501 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

7502 These functions shall compute the hyperbolic cosine of their argument *x*.

7503 An application wishing to check for error situations should set *errno* to zero and call
 7504 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 7505 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 7506 zero, an error has occurred.

7507 **RETURN VALUE**7508 Upon successful completion, these functions shall return the hyperbolic cosine of *x*.

7509 If the correct value would cause overflow, a range error shall occur and *cosh()*, *coshf()*, and
 7510 *coshl()* shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL,
 7511 respectively.

7512 **MX** If *x* is NaN, a NaN shall be returned.7513 If *x* is ± 0 , the value 1.0 shall be returned.7514 If *x* is $\pm \text{Inf}$, $\pm \text{Inf}$ shall be returned.7515 **ERRORS**

7516 These functions shall fail if:

7517 **Range Error** The result would cause an overflow.

7518 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 7519 then *errno* shall be set to [ERANGE]. If the integer expression
 7520 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow
 7521 floating-point exception shall be raised.

7522 **EXAMPLES**

7523 None.

7524 **APPLICATION USAGE**

7525 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
 7526 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

7527 For IEEE Std 754-1985 **double**, $710.5 < |x|$ implies that *cosh*(*x*) has overflowed.7528 **RATIONALE**

7529 None.

7530 **FUTURE DIRECTIONS**

7531 None.

7532 **SEE ALSO**

7533 *acosh()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, *sinh()*, *tanh()*, the Base Definitions volume of
7534 IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
7535 **<math.h>**

7536 **CHANGE HISTORY**

7537 First released in Issue 1. Derived from Issue 1 of the SVID.

7538 **Issue 5**

7539 The DESCRIPTION is updated to indicate how an application should check for an error. This
7540 text was previously published in the APPLICATION USAGE section.

7541 **Issue 6**

7542 The *coshf()* and *coshl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

7543 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
7544 revised to align with the ISO/IEC 9899:1999 standard.

7545 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
7546 marked.

7547 **NAME**

7548 cosl — cosine function

7549 **SYNOPSIS**

7550 #include <math.h>

7551 long double cosl(long double x);

7552 **DESCRIPTION**7553 Refer to *cos()*.

7554 **NAME**

7555 cpow, cpowf, cpowl — complex power functions

7556 **SYNOPSIS**

7557 #include <complex.h>

7558 double complex cpow(double complex x, double complex y);

7559 float complex cpowf(float complex x, float complex y);

7560 long double complex cpowl(long double complex x,

7561 long double complex y);

7562 **DESCRIPTION**

7563 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
7564 conflict between the requirements described here and the ISO C standard is unintentional. This
7565 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

7566 These functions shall compute the complex power function x^y , with a branch cut for the first
7567 parameter along the negative real axis.

7568 **RETURN VALUE**

7569 These functions shall return the complex power function value.

7570 **ERRORS**

7571 No errors are defined.

7572 **EXAMPLES**

7573 None.

7574 **APPLICATION USAGE**

7575 None.

7576 **RATIONALE**

7577 None.

7578 **FUTURE DIRECTIONS**

7579 None.

7580 **SEE ALSO**

7581 cabs(), csqrt(), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

7582 **CHANGE HISTORY**

7583 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

7584 **NAME**

7585 cproj, cprojf, cprojl — complex projection functions

7586 **SYNOPSIS**

7587 #include <complex.h>

7588 double complex cproj(double complex z);

7589 float complex cprojf(float complex z);

7590 long double complex cprojl(long double complex z);

7591 **DESCRIPTION**

7592 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
7593 conflict between the requirements described here and the ISO C standard is unintentional. This
7594 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

7595 These functions shall compute a projection of *z* onto the Riemann sphere: *z* projects to *z*, except
7596 that all complex infinities (even those with one infinite part and one NaN part) project to
7597 positive infinity on the real axis. If *z* has an infinite part, then *cproj(z)* shall be equivalent to:

7598 INFINITY + I * copysign(0.0, cimag(z))

7599 **RETURN VALUE**

7600 These functions shall return the value of the projection onto the Riemann sphere.

7601 **ERRORS**

7602 No errors are defined.

7603 **EXAMPLES**

7604 None.

7605 **APPLICATION USAGE**

7606 None.

7607 **RATIONALE**

7608 Two topologies are commonly used in complex mathematics: the complex plane with its
7609 continuum of infinities, and the Riemann sphere with its single infinity. The complex plane is
7610 better suited for transcendental functions, the Riemann sphere for algebraic functions. The
7611 complex types with their multiplicity of infinities provide a useful (though imperfect) model for
7612 the complex plane. The *cproj()* function helps model the Riemann sphere by mapping all
7613 infinities to one, and should be used just before any operation, especially comparisons, that
7614 might give spurious results for any of the other infinities. Note that a complex value with one
7615 infinite part and one NaN part is regarded as an infinity, not a NaN, because if one part is
7616 infinite, the complex value is infinite independent of the value of the other part. For the same
7617 reason, *cabs()* returns an infinity if its argument has an infinite part and a NaN part.

7618 **FUTURE DIRECTIONS**

7619 None.

7620 **SEE ALSO**7621 *carg()*, *cimag()*, *conj()*, *creal()*, the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>7622 **CHANGE HISTORY**

7623 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

7624 **NAME**

7625 creal, crealf, creall — complex real functions

7626 **SYNOPSIS**

7627 #include <complex.h>

7628 double creal(double complex z);

7629 float crealf(float complex z);

7630 long double creall(long double complex z);

7631 **DESCRIPTION**

7632 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
7633 conflict between the requirements described here and the ISO C standard is unintentional. This
7634 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

7635 These functions shall compute the real part of *z*.7636 **RETURN VALUE**

7637 These functions shall return the real part value.

7638 **ERRORS**

7639 No errors are defined.

7640 **EXAMPLES**

7641 None.

7642 **APPLICATION USAGE**7643 For a variable *z* of type **complex**:7644 *z* == creal(*z*) + cimag(*z*)*I7645 **RATIONALE**

7646 None.

7647 **FUTURE DIRECTIONS**

7648 None.

7649 **SEE ALSO**

7650 *carg()*, *cimag()*, *conj()*, *cproj()*, the Base Definitions volume of IEEE Std 1003.1-2001,
7651 <complex.h>

7652 **CHANGE HISTORY**

7653 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

7654 **NAME**

7655 creat — create a new file or rewrite an existing one

7656 **SYNOPSIS**

7657 OH #include <sys/stat.h>

7658 #include <fcntl.h>

7659 int creat(const char *path, mode_t mode);

7660 **DESCRIPTION**

7661 The function call:

7662 creat(path, mode)

7663 shall be equivalent to:

7664 open(path, O_WRONLY|O_CREAT|O_TRUNC, mode)

7665 **RETURN VALUE**7666 Refer to *open()*.7667 **ERRORS**7668 Refer to *open()*.7669 **EXAMPLES**7670 **Creating a File**7671 The following example creates the file **/tmp/file** with read and write permissions for the file owner and read permission for group and others. The resulting file descriptor is assigned to the *fd* variable.

7674 #include <fcntl.h>

7675 ...

7676 int fd;

7677 mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;

7678 char *filename = "/tmp/file";

7679 ...

7680 fd = creat(filename, mode);

7681 ...

7682 **APPLICATION USAGE**

7683 None.

7684 **RATIONALE**7685 The *creat()* function is redundant. Its services are also provided by the *open()* function. It has been included primarily for historical purposes since many existing applications depend on it. It is best considered a part of the C binding rather than a function that should be provided in other languages.7689 **FUTURE DIRECTIONS**

7690 None.

7691 **SEE ALSO**7692 *open()*, the Base Definitions volume of IEEE Std 1003.1-2001, <fcntl.h>, <sys/stat.h>, <sys/types.h>

7694 **CHANGE HISTORY**

7695 First released in Issue 1. Derived from Issue 1 of the SVID.

7696 **Issue 6**7697 In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.7698 The following new requirements on POSIX implementations derive from alignment with the
7699 Single UNIX Specification:

- 7700 • The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was
7701 required for conforming implementations of previous POSIX specifications, it was not
7702 required for UNIX applications.

7703 **NAME**7704 crypt — string encoding function (**CRYPT**)7705 **SYNOPSIS**7706 xSI `#include <unistd.h>`7707 `char *crypt(const char *key, const char *salt);`

7708

7709 **DESCRIPTION**7710 The *crypt()* function is a string encoding function. The algorithm is implementation-defined.7711 The *key* argument points to a string to be encoded. The *salt* argument is a string chosen from the
7712 set:

7713 a b c d e f g h i j k l m n o p q r s t u v w x y z

7714 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

7715 0 1 2 3 4 5 6 7 8 9 . /

7716 The first two characters of this string may be used to perturb the encoding algorithm.

7717 The return value of *crypt()* points to static data that is overwritten by each call.7718 The *crypt()* function need not be reentrant. A function that is not required to be reentrant is not
7719 required to be thread-safe.7720 **RETURN VALUE**7721 Upon successful completion, *crypt()* shall return a pointer to the encoded string. The first two
7722 characters of the returned value shall be those of the *salt* argument. Otherwise, it shall return a
7723 null pointer and set *errno* to indicate the error.7724 **ERRORS**7725 The *crypt()* function shall fail if:

7726 [ENOSYS] The functionality is not supported on this implementation.

7727 **EXAMPLES**7728 **Encoding Passwords**7729 The following example finds a user database entry matching a particular user name and changes
7730 the current password to a new password. The *crypt()* function generates an encoded version of
7731 each password. The first call to *crypt()* produces an encoded version of the old password; that
7732 encoded password is then compared to the password stored in the user database. The second
7733 call to *crypt()* encodes the new password before it is stored.7734 The *putpwent()* function, used in the following example, is not part of IEEE Std 1003.1-2001.7735 `#include <unistd.h>`7736 `#include <pwd.h>`7737 `#include <string.h>`7738 `#include <stdio.h>`7739 `...`7740 `int valid_change;`7741 `int pfd; /* Integer for file descriptor returned by open(). */`7742 `FILE *fpfd; /* File pointer for use in putpwent(). */`7743 `struct passwd *p;`7744 `char user[100];`7745 `char oldpasswd[100];`7746 `char newpasswd[100];`


```

7747     char savepasswd[100];
7748     ...
7749     valid_change = 0;
7750     while ((p = getpwent()) != NULL) {
7751         /* Change entry if found. */
7752         if (strcmp(p->pw_name, user) == 0) {
7753             if (strcmp(p->pw_passwd, crypt(oldpasswd, p->pw_passwd)) == 0) {
7754                 strcpy(savepasswd, crypt(newpasswd, user));
7755                 p->pw_passwd = savepasswd;
7756                 valid_change = 1;
7757             }
7758             else {
7759                 fprintf(stderr, "Old password is not valid\n");
7760             }
7761         }
7762         /* Put passwd entry into ptmp. */
7763         putpwent(p, fpfd);
7764     }

```

7765 APPLICATION USAGE

7766 The values returned by this function need not be portable among XSI-conformant systems.

7767 RATIONALE

7768 None.

7769 FUTURE DIRECTIONS

7770 None.

7771 SEE ALSO

7772 *encrypt()*, *setkey()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**unistd.h**>

7773 CHANGE HISTORY

7774 First released in Issue 1. Derived from Issue 1 of the SVID.

7775 Issue 5

7776 Normative text previously in the APPLICATION USAGE section is moved to the
7777 DESCRIPTION.

7778 **NAME**

7779 csin, csinf, csinl — complex sine functions

7780 **SYNOPSIS**

7781 #include <complex.h>

7782 double complex csin(double complex z);

7783 float complex csinf(float complex z);

7784 long double complex csinl(long double complex z);

7785 **DESCRIPTION**

7786 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
7787 conflict between the requirements described here and the ISO C standard is unintentional. This
7788 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

7789 These functions shall compute the complex sine of z.

7790 **RETURN VALUE**

7791 These functions shall return the complex sine value.

7792 **ERRORS**

7793 No errors are defined.

7794 **EXAMPLES**

7795 None.

7796 **APPLICATION USAGE**

7797 None.

7798 **RATIONALE**

7799 None.

7800 **FUTURE DIRECTIONS**

7801 None.

7802 **SEE ALSO**

7803 casin(), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

7804 **CHANGE HISTORY**

7805 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

7806 **NAME**

7807 csinh, csinhf, csinhl — complex hyperbolic sine functions

7808 **SYNOPSIS**

7809 #include <complex.h>

7810 double complex csinh(double complex z);

7811 float complex csinhf(float complex z);

7812 long double complex csinhl(long double complex z);

7813 **DESCRIPTION**

7814 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
7815 conflict between the requirements described here and the ISO C standard is unintentional. This
7816 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

7817 These functions shall compute the complex hyperbolic sine of z.

7818 **RETURN VALUE**

7819 These functions shall return the complex hyperbolic sine value.

7820 **ERRORS**

7821 No errors are defined.

7822 **EXAMPLES**

7823 None.

7824 **APPLICATION USAGE**

7825 None.

7826 **RATIONALE**

7827 None.

7828 **FUTURE DIRECTIONS**

7829 None.

7830 **SEE ALSO**7831 *casinh()*, the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>7832 **CHANGE HISTORY**

7833 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

7834 **NAME**

7835 csinl — complex sine functions

7836 **SYNOPSIS**

7837 #include <complex.h>

7838 long double complex csinl(long double complex *z*);7839 **DESCRIPTION**7840 Refer to *csin()*.

7841 **NAME**

7842 csqrt, csqrtf, csqrtl — complex square root functions

7843 **SYNOPSIS**

7844 #include <complex.h>

7845 double complex csqrt(double complex z);

7846 float complex csqrtf(float complex z);

7847 long double complex csqrtl(long double complex z);

7848 **DESCRIPTION**

7849 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
7850 conflict between the requirements described here and the ISO C standard is unintentional. This
7851 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

7852 These functions shall compute the complex square root of *z*, with a branch cut along the
7853 negative real axis.

7854 **RETURN VALUE**

7855 These functions shall return the complex square root value, in the range of the right half-plane
7856 (including the imaginary axis).

7857 **ERRORS**

7858 No errors are defined.

7859 **EXAMPLES**

7860 None.

7861 **APPLICATION USAGE**

7862 None.

7863 **RATIONALE**

7864 None.

7865 **FUTURE DIRECTIONS**

7866 None.

7867 **SEE ALSO**7868 *cabs()*, *cpow()*, the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>7869 **CHANGE HISTORY**

7870 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

7871 **NAME**

7872 ctan, ctanf, ctanl — complex tangent functions

7873 **SYNOPSIS**

7874 #include <complex.h>

7875 double complex ctan(double complex z);

7876 float complex ctanf(float complex z);

7877 long double complex ctanl(long double complex z);

7878 **DESCRIPTION**

7879 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
7880 conflict between the requirements described here and the ISO C standard is unintentional. This
7881 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

7882 These functions shall compute the complex tangent of z.

7883 **RETURN VALUE**

7884 These functions shall return the complex tangent value.

7885 **ERRORS**

7886 No errors are defined.

7887 **EXAMPLES**

7888 None.

7889 **APPLICATION USAGE**

7890 None.

7891 **RATIONALE**

7892 None.

7893 **FUTURE DIRECTIONS**

7894 None.

7895 **SEE ALSO**7896 *catan()*, the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>7897 **CHANGE HISTORY**

7898 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

7899 **NAME**

7900 ctanh, ctanhf, ctanhl — complex hyperbolic tangent functions

7901 **SYNOPSIS**

7902 #include <complex.h>

7903 double complex ctanh(double complex z);

7904 float complex ctanhf(float complex z);

7905 long double complex ctanhl(long double complex z);

7906 **DESCRIPTION**

7907 cx The functionality described on this reference page is aligned with the ISO C standard. Any
7908 conflict between the requirements described here and the ISO C standard is unintentional. This
7909 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

7910 These functions shall compute the complex hyperbolic tangent of z.

7911 **RETURN VALUE**

7912 These functions shall return the complex hyperbolic tangent value.

7913 **ERRORS**

7914 No errors are defined.

7915 **EXAMPLES**

7916 None.

7917 **APPLICATION USAGE**

7918 None.

7919 **RATIONALE**

7920 None.

7921 **FUTURE DIRECTIONS**

7922 None.

7923 **SEE ALSO**7924 *catanh()*, the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>7925 **CHANGE HISTORY**

7926 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

7927 **NAME**

7928 ctanl — complex tangent functions

7929 **SYNOPSIS**

7930 #include <complex.h>

7931 long double complex ctanl(long double complex *z*);7932 **DESCRIPTION**7933 Refer to *ctan()*.

7934 **NAME**

7935 ctermid — generate a pathname for the controlling terminal

7936 **SYNOPSIS**7937 **CX** #include <stdio.h>

7938 char *ctermid(char *s);

7939

7940 **DESCRIPTION**

7941 The *ctermid()* function shall generate a string that, when used as a pathname, refers to the
 7942 current controlling terminal for the current process. If *ctermid()* returns a pathname, access to the
 7943 file is not guaranteed.

7944 If the application uses any of the `_POSIX_THREAD_SAFE_FUNCTIONS` or `_POSIX_THREADS`
 7945 functions, it shall ensure that the *ctermid()* function is called with a non-NULL parameter.

7946 **RETURN VALUE**

7947 If *s* is a null pointer, the string shall be generated in an area that may be static (and therefore may
 7948 be overwritten by each call), the address of which shall be returned. Otherwise, *s* is assumed to
 7949 point to a character array of at least `L_ctermid` bytes; the string is placed in this array and the
 7950 value of *s* shall be returned. The symbolic constant `L_ctermid` is defined in <stdio.h>, and shall
 7951 have a value greater than 0.

7952 The *ctermid()* function shall return an empty string if the pathname that would refer to the
 7953 controlling terminal cannot be determined, or if the function is unsuccessful.

7954 **ERRORS**

7955 No errors are defined.

7956 **EXAMPLES**7957 **Determining the Controlling Terminal for the Current Process**

7958 The following example returns a pointer to a string that identifies the controlling terminal for the
 7959 current process. The pathname for the terminal is stored in the array pointed to by the *ptr*
 7960 argument, which has a size of `L_ctermid` bytes, as indicated by the *term* argument.

```
7961       #include <stdio.h>
7962       ...
7963       char term[L_ctermid];
7964       char *ptr;
7965       ptr = ctermid(term);
```

7966 **APPLICATION USAGE**

7967 The difference between *ctermid()* and *ttyname()* is that *ttyname()* must be handed a file
 7968 descriptor and return a path of the terminal associated with that file descriptor, while *ctermid()*
 7969 returns a string (such as `"/dev/tty"`) that refers to the current controlling terminal if used as a
 7970 pathname.

7971 **RATIONALE**

7972 `L_ctermid` must be defined appropriately for a given implementation and must be greater than
 7973 zero so that array declarations using it are accepted by the compiler. The value includes the
 7974 terminating null byte.

7975 Conforming applications that use threads cannot call *ctermid()* with NULL as the parameter if
 7976 either `_POSIX_THREAD_SAFE_FUNCTIONS` or `_POSIX_THREADS` is defined. If *s* is not
 7977 NULL, the *ctermid()* function generates a string that, when used as a pathname, refers to the

7978 current controlling terminal for the current process. If *s* is NULL, the return value of *ctermid()* is
7979 undefined.

7980 There is no additional burden on the programmer—changing to use a hypothetical thread-safe
7981 version of *ctermid()* along with allocating a buffer is more of a burden than merely allocating a
7982 buffer. Application code should not assume that the returned string is short, as some
7983 implementations have more than two pathname components before reaching a logical device
7984 name.

7985 **FUTURE DIRECTIONS**

7986 None.

7987 **SEE ALSO**

7988 *ttyname()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**stdio.h**>

7989 **CHANGE HISTORY**

7990 First released in Issue 1. Derived from Issue 1 of the SVID.

7991 **Issue 5**

7992 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

7993 **Issue 6**

7994 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

7995 **NAME**7996 `ctime`, `ctime_r` — convert a time value to a date and time string7997 **SYNOPSIS**7998 `#include <time.h>`7999 `char *ctime(const time_t *clock);`8000 TSF `char *ctime_r(const time_t *clock, char *buf);`

8001

8002 **DESCRIPTION**8003 CX For `ctime()`: The functionality described on this reference page is aligned with the ISO C
8004 standard. Any conflict between the requirements described here and the ISO C standard is
8005 unintentional. This volume of IEEE Std 1003.1-2001 defers to the ISO C standard.8006 The `ctime()` function shall convert the time pointed to by `clock`, representing time in seconds
8007 since the Epoch, to local time in the form of a string. It shall be equivalent to:8008 `asctime(localtime(clock))`8009 CX The `asctime()`, `ctime()`, `gmtime()`, and `localtime()` functions shall return values in one of two static
8010 objects: a broken-down time structure and an array of **char**. Execution of any of the functions
8011 may overwrite the information returned in either of these objects by any of the other functions.8012 The `ctime()` function need not be reentrant. A function that is not required to be reentrant is not
8013 required to be thread-safe.8014 TSF The `ctime_r()` function shall convert the calendar time pointed to by `clock` to local time in exactly
8015 the same form as `ctime()` and put the string into the array pointed to by `buf` (which shall be at
8016 least 26 bytes in size) and return `buf`.8017 Unlike `ctime()`, the thread-safe version `ctime_r()` is not required to set `tzname`.8018 **RETURN VALUE**8019 The `ctime()` function shall return the pointer returned by `asctime()` with that broken-down time
8020 as an argument.8021 TSF Upon successful completion, `ctime_r()` shall return a pointer to the string pointed to by `buf`.
8022 When an error is encountered, a null pointer shall be returned.8023 **ERRORS**

8024 No errors are defined.

8025 **EXAMPLES**

8026 None.

8027 **APPLICATION USAGE**8028 Values for the broken-down time structure can be obtained by calling `gmtime()` or `localtime()`.
8029 The `ctime()` function is included for compatibility with older implementations, and does not
8030 support localized date and time formats. Applications should use the `strftime()` function to
8031 achieve maximum portability.8032 The `ctime_r()` function is thread-safe and shall return values in a user-supplied buffer instead of
8033 possibly using a static data area that may be overwritten by each call.8034 **RATIONALE**

8035 None.

8036 **FUTURE DIRECTIONS**

8037 None.

8038 **SEE ALSO**

8039 *asctime()*, *clock()*, *difftime()*, *gmtime()*, *localtime()*, *mktime()*, *strftime()*, *strptime()*, *time()*, *utime()*,
8040 the Base Definitions volume of IEEE Std 1003.1-2001, <**time.h**>

8041 **CHANGE HISTORY**

8042 First released in Issue 1. Derived from Issue 1 of the SVID.

8043 **Issue 5**

8044 Normative text previously in the APPLICATION USAGE section is moved to the
8045 DESCRIPTION.

8046 The *ctime_r()* function is included for alignment with the POSIX Threads Extension.8047 A note indicating that the *ctime()* function need not be reentrant is added to the DESCRIPTION.8048 **Issue 6**

8049 Extensions beyond the ISO C standard are marked.

8050 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

8051 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
8052 its avoidance of possibly using a static data area.

8053 **NAME**

8054 daylight — daylight savings time flag

8055 **SYNOPSIS**8056 XSI `#include <time.h>`8057 `extern int daylight;`

8058

8059 **DESCRIPTION**8060 Refer to `tzset()`.

8061 NAME

8062 dbm_clearerr, dbm_close, dbm_delete, dbm_error, dbm_fetch, dbm_firstkey, dbm_nextkey,
8063 dbm_open, dbm_store — database functions

8064 SYNOPSIS

```
8065 xSI    #include <ndbm.h>

8066        int dbm_clearerr(DBM *db);
8067        void dbm_close(DBM *db);
8068        int dbm_delete(DBM *db, datum key);
8069        int dbm_error(DBM *db);
8070        datum dbm_fetch(DBM *db, datum key);
8071        datum dbm_firstkey(DBM *db);
8072        datum dbm_nextkey(DBM *db);
8073        DBM *dbm_open(const char *file, int open_flags, mode_t file_mode);
8074        int dbm_store(DBM *db, datum key, datum content, int store_mode);
8075
```

8076 DESCRIPTION

8077 These functions create, access, and modify a database.

8078 A **datum** consists of at least two members, *dptr* and *dsize*. The *dptr* member points to an object
8079 that is *dsize* bytes in length. Arbitrary binary data, as well as character strings, may be stored in
8080 the object pointed to by *dptr*.

8081 The database is stored in two files. One file is a directory containing a bitmap of keys and has
8082 **.dir** as its suffix. The second file contains all data and has **.pag** as its suffix.

8083 The *dbm_open()* function shall open a database. The *file* argument to the function is the
8084 pathname of the database. The function opens two files named *file.dir* and *file.pag*. The
8085 *open_flags* argument has the same meaning as the *flags* argument of *open()* except that a database
8086 opened for write-only access opens the files for read and write access and the behavior of the
8087 O_APPEND flag is unspecified. The *file_mode* argument has the same meaning as the third
8088 argument of *open()*.

8089 The *dbm_close()* function shall close a database. The application shall ensure that argument *db* is
8090 a pointer to a **dbm** structure that has been returned from a call to *dbm_open()*.

8091 These database functions shall support an internal block size large enough to support
8092 key/content pairs of at least 1023 bytes.

8093 The *dbm_fetch()* function shall read a record from a database. The argument *db* is a pointer to a
8094 database structure that has been returned from a call to *dbm_open()*. The argument *key* is a
8095 **datum** that has been initialized by the application to the value of the key that matches the key of
8096 the record the program is fetching.

8097 The *dbm_store()* function shall write a record to a database. The argument *db* is a pointer to a
8098 database structure that has been returned from a call to *dbm_open()*. The argument *key* is a
8099 **datum** that has been initialized by the application to the value of the key that identifies (for
8100 subsequent reading, writing, or deleting) the record the application is writing. The argument
8101 *content* is a **datum** that has been initialized by the application to the value of the record the
8102 program is writing. The argument *store_mode* controls whether *dbm_store()* replaces any pre-
8103 existing record that has the same key that is specified by the *key* argument. The application shall
8104 set *store_mode* to either DBM_INSERT or DBM_REPLACE. If the database contains a record that
8105 matches the *key* argument and *store_mode* is DBM_REPLACE, the existing record shall be
8106 replaced with the new record. If the database contains a record that matches the *key* argument
8107 and *store_mode* is DBM_INSERT, the existing record shall be left unchanged and the new record

8108 ignored. If the database does not contain a record that matches the *key* argument and *store_mode*
8109 is either DBM_INSERT or DBM_REPLACE, the new record shall be inserted in the database.

8110 If the sum of a key/content pair exceeds the internal block size, the result is unspecified.
8111 Moreover, the application shall ensure that all key/content pairs that hash together fit on a
8112 single block. The *dbm_store()* function shall return an error in the event that a disk block fills
8113 with inseparable data.

8114 The *dbm_delete()* function shall delete a record and its key from the database. The argument *db* is
8115 a pointer to a database structure that has been returned from a call to *dbm_open()*. The argument
8116 *key* is a **datum** that has been initialized by the application to the value of the key that identifies
8117 the record the program is deleting.

8118 The *dbm_firstkey()* function shall return the first key in the database. The argument *db* is a
8119 pointer to a database structure that has been returned from a call to *dbm_open()*.

8120 The *dbm_nextkey()* function shall return the next key in the database. The argument *db* is a
8121 pointer to a database structure that has been returned from a call to *dbm_open()*. The application
8122 shall ensure that the *dbm_firstkey()* function is called before calling *dbm_nextkey()*. Subsequent
8123 calls to *dbm_nextkey()* return the next key until all of the keys in the database have been
8124 returned.

8125 The *dbm_error()* function shall return the error condition of the database. The argument *db* is a
8126 pointer to a database structure that has been returned from a call to *dbm_open()*.

8127 The *dbm_clearerr()* function shall clear the error condition of the database. The argument *db* is a
8128 pointer to a database structure that has been returned from a call to *dbm_open()*.

8129 The *dptr* pointers returned by these functions may point into static storage that may be changed
8130 by subsequent calls.

8131 These functions need not be reentrant. A function that is not required to be reentrant is not
8132 required to be thread-safe.

8133 **RETURN VALUE**

8134 The *dbm_store()* and *dbm_delete()* functions shall return 0 when they succeed and a negative
8135 value when they fail.

8136 The *dbm_store()* function shall return 1 if it is called with a *flags* value of DBM_INSERT and the
8137 function finds an existing record with the same key.

8138 The *dbm_error()* function shall return 0 if the error condition is not set and return a non-zero
8139 value if the error condition is set.

8140 The return value of *dbm_clearerr()* is unspecified.

8141 The *dbm_firstkey()* and *dbm_nextkey()* functions shall return a key **datum**. When the end of the
8142 database is reached, the *dptr* member of the key is a null pointer. If an error is detected, the *dptr*
8143 member of the key shall be a null pointer and the error condition of the database shall be set.

8144 The *dbm_fetch()* function shall return a content **datum**. If no record in the database matches the
8145 key or if an error condition has been detected in the database, the *dptr* member of the content
8146 shall be a null pointer.

8147 The *dbm_open()* function shall return a pointer to a database structure. If an error is detected
8148 during the operation, *dbm_open()* shall return a (**DBM ***)0.

8149 **ERRORS**

8150 No errors are defined.

8151 **EXAMPLES**

8152 None.

8153 **APPLICATION USAGE**

8154 The following code can be used to traverse the database:

8155

```
for(key = dbm_firstkey(db); key.dptr != NULL; key = dbm_nextkey(db))
```

8156 The *dbm_** functions provided in this library should not be confused in any way with those of a
8157 general-purpose database management system. These functions do not provide for multiple
8158 search keys per entry, they do not protect against multi-user access (in other words they do not
8159 lock records or files), and they do not provide the many other useful database functions that are
8160 found in more robust database management systems. Creating and updating databases by use of
8161 these functions is relatively slow because of data copies that occur upon hash collisions. These
8162 functions are useful for applications requiring fast lookup of relatively static information that is
8163 to be indexed by a single key.

8164 Note that a strictly conforming application is extremely limited by these functions: since there is
8165 no way to determine that the keys in use do not all hash to the same value (although that would
8166 be rare), a strictly conforming application cannot be guaranteed that it can store more than one
8167 block's worth of data in the database. As long as a key collision does not occur, additional data
8168 may be stored, but because there is no way to determine whether an error is due to a key
8169 collision or some other error condition (*dbm_error()* being effectively a Boolean), once an error is
8170 detected, the application is effectively limited to guessing what the error might be if it wishes to
8171 continue using these functions.

8172 The *dbm_delete()* function need not physically reclaim file space, although it does make it
8173 available for reuse by the database.

8174 After calling *dbm_store()* or *dbm_delete()* during a pass through the keys by *dbm_firstkey()* and
8175 *dbm_nextkey()*, the application should reset the database by calling *dbm_firstkey()* before again
8176 calling *dbm_nextkey()*. The contents of these files are unspecified and may not be portable.

8177 **RATIONALE**

8178 None.

8179 **FUTURE DIRECTIONS**

8180 None.

8181 **SEE ALSO**8182 *open()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**ndbm.h**>8183 **CHANGE HISTORY**

8184 First released in Issue 4, Version 2.

8185 **Issue 5**

8186 Moved from X/OPEN UNIX extension to BASE.

8187 Normative text previously in the APPLICATION USAGE section is moved to the
8188 DESCRIPTION.

8189 A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

8190 **Issue 6**

8191 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

8192 **NAME**

8193 difftime — compute the difference between two calendar time values

8194 **SYNOPSIS**

8195 #include <time.h>

8196 double difftime(time_t *time1*, time_t *time0*);8197 **DESCRIPTION**

8198 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
8199 conflict between the requirements described here and the ISO C standard is unintentional. This
8200 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

8201 The *difftime()* function shall compute the difference between two calendar times (as returned by
8202 *time()*): *time1* − *time0*.

8203 **RETURN VALUE**8204 The *difftime()* function shall return the difference expressed in seconds as a type **double**.8205 **ERRORS**

8206 No errors are defined.

8207 **EXAMPLES**

8208 None.

8209 **APPLICATION USAGE**

8210 None.

8211 **RATIONALE**

8212 None.

8213 **FUTURE DIRECTIONS**

8214 None.

8215 **SEE ALSO**

8216 *asctime()*, *clock()*, *ctime()*, *gmtime()*, *localtime()*, *mktime()*, *strftime()*, *strptime()*, *time()*, *utime()*,
8217 the Base Definitions volume of IEEE Std 1003.1-2001, <**time.h**>

8218 **CHANGE HISTORY**

8219 First released in Issue 4. Derived from the ISO C standard.

8220 **NAME**

8221 dirname — report the parent directory name of a file pathname

8222 **SYNOPSIS**

8223 XSI #include <libgen.h>

8224 char *dirname(char *path);

8225

8226 **DESCRIPTION**

8227 The *dirname()* function shall take a pointer to a character string that contains a pathname, and
 8228 return a pointer to a string that is a pathname of the parent directory of that file. Trailing '/'
 8229 characters in the path are not counted as part of the path.

8230 If *path* does not contain a '/', then *dirname()* shall return a pointer to the string ".". If *path* is a
 8231 null pointer or points to an empty string, *dirname()* shall return a pointer to the string ".".

8232 The *dirname()* function need not be reentrant. A function that is not required to be reentrant is
 8233 not required to be thread-safe.

8234 **RETURN VALUE**

8235 The *dirname()* function shall return a pointer to a string that is the parent directory of *path*. If
 8236 *path* is a null pointer or points to an empty string, a pointer to a string "." is returned.

8237 The *dirname()* function may modify the string pointed to by *path*, and may return a pointer to
 8238 static storage that may then be overwritten by subsequent calls to *dirname()*.

8239 **ERRORS**

8240 No errors are defined.

8241 **EXAMPLES**

8242 The following code fragment reads a pathname, changes the current working directory to the
 8243 parent directory, and opens the file.

```
8244 char path[PATH_MAX], *pathcopy;
8245 int fd;
8246 fgets(path, PATH_MAX, stdin);
8247 pathcopy = strdup(path);
8248 chdir(dirname(pathcopy));
8249 fd = open(basename(path), O_RDONLY);
```

8250 **Sample Input and Output Strings for dirname()**

8251 In the following table, the input string is the value pointed to by *path*, and the output string is
 8252 the return value of the *dirname()* function.

Input String	Output String
"/usr/lib"	"/usr"
"/usr/"	"/"
"usr"	."
"/"	"/"
."	."
""	."

Changing the Current Directory to the Parent Directory

The following program fragment reads a pathname, changes the current working directory to the parent directory, and opens the file.

```
#include <unistd.h>
#include <limits.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <libgen.h>
...
char path[PATH_MAX], *pathcopy;
int fd;
...
fgets(path, PATH_MAX, stdin);
pathcopy = strdup(path);
chdir(dirname(pathcopy));
fd = open(basename(path), O_RDONLY);
```

APPLICATION USAGE

The *dirname()* and *basename()* functions together yield a complete pathname. The expression *dirname(path)* obtains the pathname of the directory where *basename(path)* is found.

Since the meaning of the leading *"/"* is implementation-defined, *dirname("/foo")* may return either *"/"* or *'/'* (but nothing else).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

basename(), the Base Definitions volume of IEEE Std 1003.1-2001, **<libgen.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

8295 **NAME**8296 `div` — compute the quotient and remainder of an integer division8297 **SYNOPSIS**8298 `#include <stdlib.h>`8299 `div_t div(int numer, int denom);`8300 **DESCRIPTION**

8301 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
8302 conflict between the requirements described here and the ISO C standard is unintentional. This
8303 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

8304 The `div()` function shall compute the quotient and remainder of the division of the numerator
8305 *numer* by the denominator *denom*. If the division is inexact, the resulting quotient is the integer
8306 of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be
8307 represented, the behavior is undefined; otherwise, *quot***denom*+*rem* shall equal *numer*.

8308 **RETURN VALUE**

8309 The `div()` function shall return a structure of type **div_t**, comprising both the quotient and the
8310 remainder. The structure includes the following members, in any order:

8311 `int quot; /* quotient */`8312 `int rem; /* remainder */`8313 **ERRORS**

8314 No errors are defined.

8315 **EXAMPLES**

8316 None.

8317 **APPLICATION USAGE**

8318 None.

8319 **RATIONALE**

8320 None.

8321 **FUTURE DIRECTIONS**

8322 None.

8323 **SEE ALSO**8324 `ldiv()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdlib.h>`8325 **CHANGE HISTORY**

8326 First released in Issue 4. Derived from the ISO C standard.

8327 **NAME**8328 dldclose — close a *dlopen()* object8329 **SYNOPSIS**

8330 XSI #include <dldfcn.h>

8331 int dldclose(void *handle);

8332

8333 **DESCRIPTION**8334 The *dldclose()* function shall inform the system that the object referenced by a *handle* returned
8335 from a previous *dlopen()* invocation is no longer needed by the application.8336 The use of *dldclose()* reflects a statement of intent on the part of the process, but does not create
8337 any requirement upon the implementation, such as removal of the code or symbols referenced
8338 by *handle*. Once an object has been closed using *dldclose()* an application should assume that its
8339 symbols are no longer available to *dlsym()*. All objects loaded automatically as a result of
8340 invoking *dlopen()* on the referenced object shall also be closed if this is the last reference to it.8341 Although a *dldclose()* operation is not required to remove structures from an address space,
8342 neither is an implementation prohibited from doing so. The only restriction on such a removal is
8343 that no object shall be removed to which references have been relocated, until or unless all such
8344 references are removed. For instance, an object that had been loaded with a *dlopen()* operation
8345 specifying the RTLD_GLOBAL flag might provide a target for dynamic relocations performed in
8346 the processing of other objects—in such environments, an application may assume that no
8347 relocation, once made, shall be undone or remade unless the object requiring the relocation has
8348 itself been removed.8349 **RETURN VALUE**8350 If the referenced object was successfully closed, *dldclose()* shall return 0. If the object could not be
8351 closed, or if *handle* does not refer to an open object, *dldclose()* shall return a non-zero value. More
8352 detailed diagnostic information shall be available through *dlderror()*.8353 **ERRORS**

8354 No errors are defined.

8355 **EXAMPLES**8356 The following example illustrates use of *dlopen()* and *dldclose()*:8357 ...
8358 /* Open a dynamic library and then close it ... */

8359 #include <dldfcn.h>
8360 void *mylib;
8361 int eret;

8362 mylib = dlopen("mylib.so", RTLD_LOCAL | RTLD_LAZY);
8363 ...
8364 eret = dldclose(mylib);
8365 ...8366 **APPLICATION USAGE**8367 A conforming application should employ a *handle* returned from a *dlopen()* invocation only
8368 within a given scope bracketed by the *dlopen()* and *dldclose()* operations. Implementations are
8369 free to use reference counting or other techniques such that multiple calls to *dlopen()* referencing
8370 the same object may return the same object for *handle*. Implementations are also free to reuse a
8371 *handle*. For these reasons, the value of a *handle* must be treated as an opaque object by the
8372 application, used only in calls to *dlsym()* and *dldclose()*.

8373 **RATIONALE**

8374 None.

8375 **FUTURE DIRECTIONS**

8376 None.

8377 **SEE ALSO**8378 *dLError()*, *dlopen()*, *dlsym()*, the Base Definitions volume of IEEE Std 1003.1-2001, <dlfcn.h>8379 **CHANGE HISTORY**

8380 First released in Issue 5.

8381 **Issue 6**8382 The DESCRIPTION is updated to say that the referenced object is closed “if this is the last
8383 reference to it”.

8384 NAME

8385 dlderror — get diagnostic information

8386 SYNOPSIS

8387 XSI #include <dldfcn.h>

8388 char *dlderror(void);

8389

8390 DESCRIPTION

8391 The *dlderror()* function shall return a null-terminated character string (with no trailing <newline>)
8392 that describes the last error that occurred during dynamic linking processing. If no dynamic
8393 linking errors have occurred since the last invocation of *dlderror()*, *dlderror()* shall return NULL.
8394 Thus, invoking *dlderror()* a second time, immediately following a prior invocation, shall result in
8395 NULL being returned.

8396 The *dlderror()* function need not be reentrant. A function that is not required to be reentrant is not
8397 required to be thread-safe.

8398 RETURN VALUE

8399 If successful, *dlderror()* shall return a null-terminated character string; otherwise, NULL shall be
8400 returned.

8401 ERRORS

8402 No errors are defined.

8403 EXAMPLES

8404 The following example prints out the last dynamic linking error:

```
8405 ...  
8406 #include <dldfcn.h>  
8407 char *errstr;  
8408 errstr = dlderror();  
8409 if (errstr != NULL)  
8410 printf ("A dynamic linking error occurred: (%s)\n", errstr);  
8411 ...
```

8412 APPLICATION USAGE

8413 The messages returned by *dlderror()* may reside in a static buffer that is overwritten on each call
8414 to *dlderror()*. Application code should not write to this buffer. Programs wishing to preserve an
8415 error message should make their own copies of that message. Depending on the application
8416 environment with respect to asynchronous execution events, such as signals or other
8417 asynchronous computation sharing the address space, conforming applications should use a
8418 critical section to retrieve the error pointer and buffer.

8419 RATIONALE

8420 None.

8421 FUTURE DIRECTIONS

8422 None.

8423 SEE ALSO

8424 *dldclose()*, *dldopen()*, *dldsym()*, the Base Definitions volume of IEEE Std 1003.1-2001, <dldfcn.h>

8425	CHANGE HISTORY
8426	First released in Issue 5.
8427	Issue 6
8428	In the DESCRIPTION the note about reentrancy and thread-safety is added.

8429 NAME

8430 dlopen — gain access to an executable object file

8431 SYNOPSIS

8432 xSI #include <dlfcn.h>

8433 void *dlopen(const char *file, int mode);

8434

8435 DESCRIPTION

8436 The *dlopen()* function shall make an executable object file specified by *file* available to the calling
 8437 program. The class of files eligible for this operation and the manner of their construction are
 8438 implementation-defined, though typically such files are executable objects such as shared
 8439 libraries, relocatable files, or programs. Note that some implementations permit the construction
 8440 of dependencies between such objects that are embedded within files. In such cases, a *dlopen()*
 8441 operation shall load such dependencies in addition to the object referenced by *file*.
 8442 Implementations may also impose specific constraints on the construction of programs that can
 8443 employ *dlopen()* and its related services.

8444 A successful *dlopen()* shall return a *handle* which the caller may use on subsequent calls to
 8445 *dlsym()* and *dlclose()*. The value of this *handle* should not be interpreted in any way by the caller.

8446 The *file* argument is used to construct a pathname to the object file. If *file* contains a slash
 8447 character, the *file* argument is used as the pathname for the file. Otherwise, *file* is used in an
 8448 implementation-defined manner to yield a pathname.

8449 If the value of *file* is 0, *dlopen()* shall provide a *handle* on a global symbol object. This object shall
 8450 provide access to the symbols from an ordered set of objects consisting of the original program
 8451 image file, together with any objects loaded at program start-up as specified by that process
 8452 image file (for example, shared libraries), and the set of objects loaded using a *dlopen()* operation
 8453 together with the RTLD_GLOBAL flag. As the latter set of objects can change during execution,
 8454 the set identified by *handle* can also change dynamically.

8455 Only a single copy of an object file is brought into the address space, even if *dlopen()* is invoked
 8456 multiple times in reference to the file, and even if different pathnames are used to reference the
 8457 file.

8458 The *mode* parameter describes how *dlopen()* shall operate upon *file* with respect to the processing
 8459 of relocations and the scope of visibility of the symbols provided within *file*. When an object is
 8460 brought into the address space of a process, it may contain references to symbols whose
 8461 addresses are not known until the object is loaded. These references shall be relocated before the
 8462 symbols can be accessed. The *mode* parameter governs when these relocations take place and
 8463 may have the following values:

8464 8465 8466 8467 8468 8469 8470 8471	RTLD_LAZY	Relocations shall be performed at an implementation-defined time, ranging from the time of the <i>dlopen()</i> call until the first reference to a given symbol occurs. Specifying RTLD_LAZY should improve performance on implementations supporting dynamic symbol binding as a process may not reference all of the functions in any given object. And, for systems supporting dynamic symbol resolution for normal process execution, this behavior mimics the normal handling of process execution.
--	-----------	--

8472 8473 8474 8475	RTLD_NOW	All necessary relocations shall be performed when the object is first loaded. This may waste some processing if relocations are performed for functions that are never referenced. This behavior may be useful for applications that need to know as soon as an object is loaded that all
------------------------------	----------	---

8476 symbols referenced during execution are available.

8477 Any object loaded by *dlopen()* that requires relocations against global symbols can reference the
 8478 symbols in the original process image file, any objects loaded at program start-up, from the
 8479 object itself as well as any other object included in the same *dlopen()* invocation, and any objects
 8480 that were loaded in any *dlopen()* invocation and which specified the RTLD_GLOBAL flag. To
 8481 determine the scope of visibility for the symbols loaded with a *dlopen()* invocation, the *mode*
 8482 parameter should be a bitwise-inclusive OR with one of the following values:

8483 RTLD_GLOBAL The object's symbols shall be made available for the relocation processing
 8484 of any other object. In addition, symbol lookup using *dlopen(0, mode)* and
 8485 an associated *dlsym()* allows objects loaded with this *mode* to be searched.

8486 RTLD_LOCAL The object's symbols shall not be made available for the relocation
 8487 processing of any other object.

8488 If neither RTLD_GLOBAL nor RTLD_LOCAL are specified, then an implementation-defined
 8489 default behavior shall be applied.

8490 If a *file* is specified in multiple *dlopen()* invocations, *mode* is interpreted at each invocation. Note,
 8491 however, that once RTLD_NOW has been specified all relocations shall have been completed
 8492 rendering further RTLD_NOW operations redundant and any further RTLD_LAZY operations
 8493 irrelevant. Similarly, note that once RTLD_GLOBAL has been specified the object shall maintain
 8494 the RTLD_GLOBAL status regardless of any previous or future specification of RTLD_LOCAL,
 8495 as long as the object remains in the address space (see *dlclose()*).

8496 Symbols introduced into a program through calls to *dlopen()* may be used in relocation
 8497 activities. Symbols so introduced may duplicate symbols already defined by the program or
 8498 previous *dlopen()* operations. To resolve the ambiguities such a situation might present, the
 8499 resolution of a symbol reference to symbol definition is based on a symbol resolution order. Two
 8500 such resolution orders are defined: *load* or *dependency* ordering. Load order establishes an
 8501 ordering among symbol definitions, such that the definition first loaded (including definitions
 8502 from the image file and any dependent objects loaded with it) has priority over objects added
 8503 later (via *dlopen()*). Load ordering is used in relocation processing. Dependency ordering uses a
 8504 breadth-first order starting with a given object, then all of its dependencies, then any dependents
 8505 of those, iterating until all dependencies are satisfied. With the exception of the global symbol
 8506 object obtained via a *dlopen()* operation on a *file* of 0, dependency ordering is used by the
 8507 *dlsym()* function. Load ordering is used in *dlsym()* operations upon the global symbol object.

8508 When an object is first made accessible via *dlopen()* it and its dependent objects are added in
 8509 dependency order. Once all the objects are added, relocations are performed using load order.
 8510 Note that if an object or its dependencies had been previously loaded, the load and dependency
 8511 orders may yield different resolutions.

8512 The symbols introduced by *dlopen()* operations and available through *dlsym()* are at a minimum
 8513 those which are exported as symbols of global scope by the object. Typically such symbols shall
 8514 be those that were specified in (for example) C source code as having *extern* linkage. The precise
 8515 manner in which an implementation constructs the set of exported symbols for a *dlopen()* object
 8516 is specified by that implementation.

8517 RETURN VALUE

8518 If *file* cannot be found, cannot be opened for reading, is not of an appropriate object format for
 8519 processing by *dlopen()*, or if an error occurs during the process of loading *file* or relocating its
 8520 symbolic references, *dlopen()* shall return NULL. More detailed diagnostic information shall be
 8521 available through *dlerror()*.

8522 ERRORS

8523 No errors are defined.

8524 EXAMPLES

8525 None.

8526 APPLICATION USAGE

8527 None.

8528 RATIONALE

8529 None.

8530 FUTURE DIRECTIONS

8531 None.

8532 SEE ALSO

8533 *dlclose()*, *dLError()*, *dlsym()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**dlfcn.h**>

8534 CHANGE HISTORY

8535 First released in Issue 5.

8536 **NAME**8537 dlsym — obtain the address of a symbol from a *dlopen()* object8538 **SYNOPSIS**8539 **XSI** #include <dlfcn.h>8540 void *dlsym(void *restrict *handle*, const char *restrict *name*);

8541

8542 **DESCRIPTION**

8543 The *dlsym()* function shall obtain the address of a symbol defined within an object made
 8544 accessible through a *dlopen()* call. The *handle* argument is the value returned from a call to
 8545 *dlopen()* (and which has not since been released via a call to *dlclose()*), and *name* is the symbol's
 8546 name as a character string.

8547 The *dlsym()* function shall search for the named symbol in all objects loaded automatically as a
 8548 result of loading the object referenced by *handle* (see *dlopen()*). Load ordering is used in *dlsym()*
 8549 operations upon the global symbol object. The symbol resolution algorithm used shall be
 8550 dependency order as described in *dlopen()*.

8551 The RTLD_DEFAULT and RTLD_NEXT flags are reserved for future use.

8552 **RETURN VALUE**

8553 If *handle* does not refer to a valid object opened by *dlopen()*, or if the named symbol cannot be
 8554 found within any of the objects associated with *handle*, *dlsym()* shall return NULL. More
 8555 detailed diagnostic information shall be available through *dlerror()*.

8556 **ERRORS**

8557 No errors are defined.

8558 **EXAMPLES**

8559 The following example shows how *dlopen()* and *dlsym()* can be used to access either function or
 8560 data objects. For simplicity, error checking has been omitted.

```
8561       void       *handle;
8562       int        *iptr, (*fptr)(int);

8563       /* open the needed object */
8564       handle = dlopen("/usr/home/me/libfoo.so", RTLD_LOCAL | RTLD_LAZY);

8565       /* find the address of function and data objects */
8566       fptr = (int (*)(int))dlsym(handle, "my_function");
8567       iptr = (int *)dlsym(handle, "my_object");

8568       /* invoke function, passing value of integer as a parameter */
8569       (*fptr)(*iptr);
```

8570 **APPLICATION USAGE**

8571 Special purpose values for *handle* are reserved for future use. These values and their meanings
 8572 are:

8573 RTLD_DEFAULT The symbol lookup happens in the normal global scope; that is, a search for a
 8574 symbol using this handle would find the same definition as a direct use of this
 8575 symbol in the program code.

8576 RTLD_NEXT Specifies the next object after this one that defines *name*. *This one* refers to the
 8577 object containing the invocation of *dlsym()*. The *next* object is the one found
 8578 upon the application of a load order symbol resolution algorithm (see
 8579 *dlopen()*). The next object is either one of global scope (because it was
 8580 introduced as part of the original process image or because it was added with

8581 a *dlopen()* operation including the `RTLD_GLOBAL` flag), or is an object that
8582 was included in the same *dlopen()* operation that loaded this one.

8583 The `RTLD_NEXT` flag is useful to navigate an intentionally created hierarchy
8584 of multiply-defined symbols created through *interposition*. For example, if a
8585 program wished to create an implementation of *malloc()* that embedded some
8586 statistics gathering about memory allocations, such an implementation could
8587 use the real *malloc()* definition to perform the memory allocation—and itself
8588 only embed the necessary logic to implement the statistics gathering function.

8589 **RATIONALE**
8590 None.

8591 **FUTURE DIRECTIONS**
8592 None.

8593 **SEE ALSO**
8594 *dlclose()*, *dlderror()*, *dlopen()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**dlfcn.h**>

8595 **CHANGE HISTORY**
8596 First released in Issue 5.

8597 **Issue 6**
8598 The **restrict** keyword is added to the *dlsym()* prototype for alignment with the
8599 ISO/IEC 9899:1999 standard.

8600 The `RTLD_DEFAULT` and `RTLD_NEXT` flags are reserved for future use.

8601 **NAME**

8602 drand48, erand48, jrand48, lcong48, lrand48, mrand48, nrand48, seed48, srand48 — generate
 8603 uniformly distributed pseudo-random numbers

8604 **SYNOPSIS**

```
8605 xSI #include <stdlib.h>

8606 double drand48(void);
8607 double erand48(unsigned short xsubi[3]);
8608 long jrand48(unsigned short xsubi[3]);
8609 void lcong48(unsigned short param[7]);
8610 long lrand48(void);
8611 long mrand48(void);
8612 long nrand48(unsigned short xsubi[3]);
8613 unsigned short *seed48(unsigned short seed16v[3]);
8614 void srand48(long seedval);
8615
```

8616 **DESCRIPTION**

8617 This family of functions shall generate pseudo-random numbers using a linear congruential
 8618 algorithm and 48-bit integer arithmetic.

8619 The *drand48()* and *erand48()* functions shall return non-negative, double-precision, floating-
 8620 point values, uniformly distributed over the interval [0.0,1.0).

8621 The *lrand48()* and *nrand48()* functions shall return non-negative, long integers, uniformly
 8622 distributed over the interval $[0, 2^{31})$.

8623 The *mrnd48()* and *jrnd48()* functions shall return signed long integers uniformly distributed
 8624 over the interval $[-2^{31}, 2^{31})$.

8625 The *srand48()*, *seed48()*, and *lcong48()* functions are initialization entry points, one of which
 8626 should be invoked before either *drand48()*, *lrand48()*, or *mrnd48()* is called. (Although it is not
 8627 recommended practice, constant default initializer values shall be supplied automatically if
 8628 *drand48()*, *lrand48()*, or *mrnd48()* is called without a prior call to an initialization entry point.)
 8629 The *erand48()*, *nrand48()*, and *jrnd48()* functions do not require an initialization entry point to
 8630 be called first.

8631 All the routines work by generating a sequence of 48-bit integer values, X_i , according to the
 8632 linear congruential formula:

$$8633 \quad X_{n+1} = (aX_n + c)_{\text{mod } m} \quad n \geq 0$$

8634 The parameter $m = 2^{48}$; hence 48-bit integer arithmetic is performed. Unless *lcong48()* is invoked,
 8635 the multiplier value a and the addend value c are given by:

$$8636 \quad a = 5\text{DEECE66D}_{16} = 273673163155_8$$

$$8637 \quad c = \text{B}_{16} = 13_8$$

8638 The value returned by any of the *drand48()*, *erand48()*, *jrnd48()*, *lrand48()*, *mrnd48()*, or
 8639 *nrand48()* functions is computed by first generating the next 48-bit X_i in the sequence. Then the
 8640 appropriate number of bits, according to the type of data item to be returned, are copied from
 8641 the high-order (leftmost) bits of X_i and transformed into the returned value.

8642 The *drand48()*, *lrand48()*, and *mrnd48()* functions store the last 48-bit X_i generated in an
 8643 internal buffer; that is why the application shall ensure that these are initialized prior to being
 8644 invoked. The *erand48()*, *nrand48()*, and *jrnd48()* functions require the calling program to
 8645 provide storage for the successive X_i values in the array specified as an argument when the

8646 functions are invoked. That is why these routines do not have to be initialized; the calling
 8647 program merely has to place the desired initial value of X_i into the array and pass it as an
 8648 argument. By using different arguments, *erand48()*, *rand48()*, and *jrand48()* allow separate
 8649 modules of a large program to generate several *independent* streams of pseudo-random numbers;
 8650 that is, the sequence of numbers in each stream shall *not* depend upon how many times the
 8651 routines are called to generate numbers for the other streams.

8652 The initializer function *srand48()* sets the high-order 32 bits of X_i to the low-order 32 bits
 8653 contained in its argument. The low-order 16 bits of X_i are set to the arbitrary value $330E_{16}$.

8654 The initializer function *seed48()* sets the value of X_i to the 48-bit value specified in the argument
 8655 array. The low-order 16 bits of X_i are set to the low-order 16 bits of *seed16v*[0]. The mid-order 16
 8656 bits of X_i are set to the low-order 16 bits of *seed16v*[1]. The high-order 16 bits of X_i are set to the
 8657 low-order 16 bits of *seed16v*[2]. In addition, the previous value of X_i is copied into a 48-bit
 8658 internal buffer, used only by *seed48()*, and a pointer to this buffer is the value returned by
 8659 *seed48()*. This returned pointer, which can just be ignored if not needed, is useful if a program is
 8660 to be restarted from a given point at some future time—use the pointer to get at and store the
 8661 last X_i value, and then use this value to reinitialize via *seed48()* when the program is restarted.

8662 The initializer function *lcg48()* allows the user to specify the initial X_i , the multiplier value *a*,
 8663 and the addend value *c*. Argument array elements *param*[0-2] specify X_i , *param*[3-5] specify the
 8664 multiplier *a*, and *param*[6] specifies the 16-bit addend *c*. After *lcg48()* is called, a subsequent
 8665 call to either *srand48()* or *seed48()* shall restore the standard multiplier and addend values, *a* and
 8666 *c*, specified above.

8667 The *drand48()*, *lrnd48()*, and *mrnd48()* functions need not be reentrant. A function that is not
 8668 required to be reentrant is not required to be thread-safe.

8669 RETURN VALUE

8670 As described in the DESCRIPTION above.

8671 ERRORS

8672 No errors are defined.

8673 EXAMPLES

8674 None.

8675 APPLICATION USAGE

8676 None.

8677 RATIONALE

8678 None.

8679 FUTURE DIRECTIONS

8680 None.

8681 SEE ALSO

8682 *rand()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

8683 CHANGE HISTORY

8684 First released in Issue 1. Derived from Issue 1 of the SVID.

8685 Issue 5

8686 A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

8687 Issue 6

8688 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

8689 **NAME**

8690 dup, dup2 — duplicate an open file descriptor

8691 **SYNOPSIS**

8692 #include <unistd.h>

8693 int dup(int *fil-des*);8694 int dup2(int *fil-des*, int *fil-des2*);8695 **DESCRIPTION**8696 The *dup()* and *dup2()* functions provide an alternative interface to the service provided by *fcntl()* using the *F_DUPFD* command. The call:8698 *fid* = dup(*fil-des*);

8699 shall be equivalent to:

8700 *fid* = fcntl(*fil-des*, *F_DUPFD*, 0);

8701 The call:

8702 *fid* = dup2(*fil-des*, *fil-des2*);

8703 shall be equivalent to:

8704 close(*fil-des2*);8705 *fid* = fcntl(*fil-des*, *F_DUPFD*, *fil-des2*);

8706 except for the following:

- 8707 • If *fil-des2* is less than 0 or greater than or equal to {*OPEN_MAX*}, *dup2()* shall return *-1* with
- 8708 *errno* set to [*EBADF*].
- 8709 • If *fil-des* is a valid file descriptor and is equal to *fil-des2*, *dup2()* shall return *fil-des2* without
- 8710 closing it.
- 8711 • If *fil-des* is not a valid file descriptor, *dup2()* shall return *-1* and shall not close *fil-des2*.
- 8712 • The value returned shall be equal to the value of *fil-des2* upon successful completion, or *-1*
- 8713 upon failure.

8714 **RETURN VALUE**

8715 Upon successful completion a non-negative integer, namely the file descriptor, shall be returned;

8716 otherwise, *-1* shall be returned and *errno* set to indicate the error.8717 **ERRORS**8718 The *dup()* function shall fail if:8719 [*EBADF*] The *fil-des* argument is not a valid open file descriptor.8720 [*EMFILE*] The number of file descriptors in use by this process would exceed8721 {*OPEN_MAX*}.8722 The *dup2()* function shall fail if:8723 [*EBADF*] The *fil-des* argument is not a valid open file descriptor or the argument *fil-des2* is8724 negative or greater than or equal to {*OPEN_MAX*}.8725 [*EINTR*] The *dup2()* function was interrupted by a signal.

8726 **EXAMPLES**8727 **Redirecting Standard Output to a File**

8728 The following example closes standard output for the current processes, re-assigns standard
8729 output to go to the file referenced by *pdf*, and closes the original file descriptor to clean up.

```
8730 #include <unistd.h>
8731 ...
8732 int pdf;
8733 ...
8734 close(1);
8735 dup(pdf);
8736 close(pdf);
8737 ...
```

8738 **Redirecting Error Messages**

8739 The following example redirects messages from *stderr* to *stdout*.

```
8740 #include <unistd.h>
8741 ...
8742 dup2(1, 2);
8743 ...
```

8744 **APPLICATION USAGE**

8745 None.

8746 **RATIONALE**

8747 The *dup()* and *dup2()* functions are redundant. Their services are also provided by the *fcntl()*
8748 function. They have been included in this volume of IEEE Std 1003.1-2001 primarily for historical
8749 reasons, since many existing applications use them.

8750 While the brief code segment shown is very similar in behavior to *dup2()*, a conforming
8751 implementation based on other functions defined in this volume of IEEE Std 1003.1-2001 is
8752 significantly more complex. Least obvious is the possible effect of a signal-catching function that
8753 could be invoked between steps and allocate or deallocate file descriptors. This could be avoided
8754 by blocking signals.

8755 The *dup2()* function is not marked obsolescent because it presents a type-safe version of
8756 functionality provided in a type-unsafe version by *fcntl()*. It is used in the POSIX Ada binding.

8757 The *dup2()* function is not intended for use in critical regions as a synchronization mechanism.

8758 In the description of [EBADF], the case of *fildes* being out of range is covered by the given case of
8759 *fildes* not being valid. The descriptions for *fildes* and *fildes2* are different because the only kind of
8760 invalidity that is relevant for *fildes2* is whether it is out of range; that is, it does not matter
8761 whether *fildes2* refers to an open file when the *dup2()* call is made.

8762 **FUTURE DIRECTIONS**

8763 None.

8764 **SEE ALSO**

8765 *close()*, *fcntl()*, *open()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**unistd.h**>

8766 **CHANGE HISTORY**

8767 First released in Issue 1. Derived from Issue 1 of the SVID.

8768 NAME

8769 ecvt, fcvt, gcvt — convert a floating-point number to a string (**LEGACY**)

8770 SYNOPSIS

8771 XSI `#include <stdlib.h>`

```

8772 char *ecvt(double value, int ndigit, int *restrict decpt,
8773           int *restrict sign);
8774 char *fcvt(double value, int ndigit, int *restrict decpt,
8775           int *restrict sign);
8776 char *gcvt(double value, int ndigit, char *buf);
8777

```

8778 DESCRIPTION

8779 The *ecvt()*, *fcvt()*, and *gcvt()* functions shall convert floating-point numbers to null-terminated strings.

8781 The *ecvt()* function shall convert *value* to a null-terminated string of *ndigit* digits (where *ndigit* is reduced to an unspecified limit determined by the precision of a **double**) and return a pointer to the string. The high-order digit shall be non-zero, unless the value is 0. The low-order digit shall be rounded in an implementation-defined manner. The position of the radix character relative to the beginning of the string shall be stored in the integer pointed to by *decpt* (negative means to the left of the returned digits). If *value* is zero, it is unspecified whether the integer pointed to by *decpt* would be 0 or 1. The radix character shall not be included in the returned string. If the sign of the result is negative, the integer pointed to by *sign* shall be non-zero; otherwise, it shall be 0.

8789 If the converted value is out of range or is not representable, the contents of the returned string are unspecified.

8791 The *fcvt()* function shall be equivalent to *ecvt()*, except that *ndigit* specifies the number of digits desired after the radix character. The total number of digits in the result string is restricted to an unspecified limit as determined by the precision of a **double**.

8794 The *gcvt()* function shall convert *value* to a null-terminated string (similar to that of the %g conversion specification format of *printf()*) in the array pointed to by *buf* and shall return *buf*. It shall produce *ndigit* significant digits (limited to an unspecified value determined by the precision of a **double**) in the %F conversion specification format of *printf()* if possible, or the %e conversion specification format of *printf()* (scientific notation) otherwise. A minus sign shall be included in the returned string if *value* is less than 0. A radix character shall be included in the returned string if *value* is not a whole number. Trailing zeros shall be suppressed where *value* is not a whole number. The radix character is determined by the current locale. If *setlocale()* has not been called successfully, the default locale, POSIX, is used. The default locale specifies a period ('.') as the radix character. The *LC_NUMERIC* category determines the value of the radix character within the current locale.

8805 These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

8807 RETURN VALUE

8808 The *ecvt()* and *fcvt()* functions shall return a pointer to a null-terminated string of digits.8809 The *gcvt()* function shall return *buf*.

8810 The return values from *ecvt()* and *fcvt()* may point to static data which may be overwritten by subsequent calls to these functions.

8812 ERRORS

8813 No errors are defined.

8814 EXAMPLES

8815 None.

8816 APPLICATION USAGE

8817 The *sprintf()* function is preferred over this function.

8818 RATIONALE

8819 None.

8820 FUTURE DIRECTIONS

8821 These functions may be withdrawn in a future version.

8822 SEE ALSO

8823 *printf()*, *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

8824 CHANGE HISTORY

8825 First released in Issue 4, Version 2.

8826 Issue 5

8827 Moved from X/OPEN UNIX extension to BASE.

8828 Normative text previously in the APPLICATION USAGE section is moved to the
8829 DESCRIPTION.

8830 A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

8831 Issue 6

8832 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

8833 This function is marked LEGACY.

8834 The **restrict** keyword is added to the *ecvt()* and *fcvt()* prototypes for alignment with the
8835 ISO/IEC 9899:1999 standard.

8836 The DESCRIPTION is updated to explicitly use “conversion specification” to describe %g, %f,
8837 and %e.

8838 **NAME**8839 encrypt — encoding function (**CRYPT**)8840 **SYNOPSIS**

8841 XSI #include <unistd.h>

8842 void encrypt(char block[64], int edflag);

8843

8844 **DESCRIPTION**

8845 The *encrypt()* function shall provide access to an implementation-defined encoding algorithm.
8846 The key generated by *setkey()* is used to encrypt the string *block* with *encrypt()*.

8847 The *block* argument to *encrypt()* shall be an array of length 64 bytes containing only the bytes
8848 with values of 0 and 1. The array is modified in place to a similar array using the key set by
8849 *setkey()*. If *edflag* is 0, the argument is encoded. If *edflag* is 1, the argument may be decoded (see
8850 the APPLICATION USAGE section); if the argument is not decoded, *errno* shall be set to
8851 [ENOSYS].

8852 The *encrypt()* function shall not change the setting of *errno* if successful. An application wishing
8853 to check for error situations should set *errno* to 0 before calling *encrypt()*. If *errno* is non-zero on
8854 return, an error has occurred.

8855 The *encrypt()* function need not be reentrant. A function that is not required to be reentrant is
8856 not required to be thread-safe.

8857 **RETURN VALUE**8858 The *encrypt()* function shall not return a value.8859 **ERRORS**8860 The *encrypt()* function shall fail if:

8861 [ENOSYS] The functionality is not supported on this implementation.

8862 **EXAMPLES**

8863 None.

8864 **APPLICATION USAGE**8865 Historical implementations of the *encrypt()* function used a rather primitive encoding algorithm.

8866 In some environments, decoding might not be implemented. This is related to some Government
8867 restrictions on encryption and decryption routines. Historical practice has been to ship a
8868 different version of the encryption library without the decryption feature in the routines
8869 supplied. Thus the exported version of *encrypt()* does encoding but not decoding.

8870 **RATIONALE**

8871 None.

8872 **FUTURE DIRECTIONS**

8873 None.

8874 **SEE ALSO**8875 *crypt()*, *setkey()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>8876 **CHANGE HISTORY**

8877 First released in Issue 1. Derived from Issue 1 of the SVID.

8878 Issue 5

8879 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

8880 Issue 6

8881 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

8882 NAME

8883 endgrent, getgrent, setgrent — group database entry functions

8884 SYNOPSIS

8885 XSI `#include <grp.h>`

```
8886 void endgrent(void);
8887 struct group *getgrent(void);
8888 void setgrent(void);
8889
```

8890 DESCRIPTION

8891 The *getgrent()* function shall return a pointer to a structure containing the broken-out fields of an
 8892 entry in the group database. When first called, *getgrent()* shall return a pointer to a **group**
 8893 structure containing the first entry in the group database. Thereafter, it shall return a pointer to a
 8894 **group** structure containing the next group structure in the group database, so successive calls
 8895 may be used to search the entire database.

8896 An implementation that provides extended security controls may impose further
 8897 implementation-defined restrictions on accessing the group database. In particular, the system
 8898 may deny the existence of some or all of the group database entries associated with groups other
 8899 than those groups associated with the caller and may omit users other than the caller from the
 8900 list of members of groups in database entries that are returned.

8901 The *setgrent()* function shall rewind the group database to allow repeated searches.

8902 The *endgrent()* function may be called to close the group database when processing is complete.

8903 These functions need not be reentrant. A function that is not required to be reentrant is not
 8904 required to be thread-safe.

8905 RETURN VALUE

8906 When first called, *getgrent()* shall return a pointer to the first group structure in the group
 8907 database. Upon subsequent calls it shall return the next group structure in the group database.
 8908 The *getgrent()* function shall return a null pointer on end-of-file or an error and *errno* may be set
 8909 to indicate the error.

8910 The return value may point to a static area which is overwritten by a subsequent call to
 8911 *getgrgid()*, *getgrnam()*, or *getgrent()*.

8912 ERRORS

8913 The *getgrent()* function may fail if:

- | | | |
|------|----------|--|
| 8914 | [EINTR] | A signal was caught during the operation. |
| 8915 | [EIO] | An I/O error has occurred. |
| 8916 | [EMFILE] | {OPEN_MAX} file descriptors are currently open in the calling process. |
| 8917 | [ENFILE] | The maximum allowable number of files is currently open in the system. |

8918 EXAMPLES

8919 None.

8920 APPLICATION USAGE

8921 These functions are provided due to their historical usage. Applications should avoid
8922 dependencies on fields in the group database, whether the database is a single file, or where in
8923 the file system name space the database resides. Applications should use *getgrnam()* and
8924 *getgrgid()* whenever possible because it avoids these dependencies.

8925 RATIONALE

8926 None.

8927 FUTURE DIRECTIONS

8928 None.

8929 SEE ALSO

8930 *getgrgid()*, *getgrnam()*, *getlogin()*, *getpwent()*, the Base Definitions volume of
8931 IEEE Std 1003.1-2001, <grp.h>

8932 CHANGE HISTORY

8933 First released in Issue 4, Version 2.

8934 Issue 5

8935 Moved from X/OPEN UNIX extension to BASE.

8936 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
8937 VALUE section.

8938 A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

8939 Issue 6

8940 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

8941 NAME

8942 endhostent, gethostent, sethostent — network host database functions

8943 SYNOPSIS

8944 #include <netdb.h>

8945 void endhostent(void);

8946 struct hostent *gethostent(void);

8947 void sethostent(int stayopen);

8948 DESCRIPTION

8949 These functions shall retrieve information about hosts. This information is considered to be
8950 stored in a database that can be accessed sequentially or randomly. The implementation of this
8951 database is unspecified.

8952 **Note:** In many cases this database is implemented by the Domain Name System, as documented in
8953 RFC 1034, RFC 1035, and RFC 1886.

8954 The *sethostent()* function shall open a connection to the database and set the next entry for
8955 retrieval to the first entry in the database. If the *stayopen* argument is non-zero, the connection
8956 shall not be closed by a call to *gethostent()*, *gethostbyname()*, or *gethostbyaddr()*, and the
8957 implementation may maintain an open file descriptor.

8958 The *gethostent()* function shall read the next entry in the database, opening and closing a
8959 connection to the database as necessary.

8960 Entries shall be returned in **hostent** structures. Refer to *gethostbyaddr()* for a definition of the
8961 **hostent** structure.

8962 The *endhostent()* function shall close the connection to the database, releasing any open file
8963 descriptor.

8964 These functions need not be reentrant. A function that is not required to be reentrant is not
8965 required to be thread-safe.

8966 RETURN VALUE

8967 Upon successful completion, the *gethostent()* function shall return a pointer to a **hostent**
8968 structure if the requested entry was found, and a null pointer if the end of the database was
8969 reached or the requested entry was not found.

8970 ERRORS

8971 No errors are defined for *endhostent()*, *gethostent()*, and *sethostent()*.

8972 EXAMPLES

8973 None.

8974 APPLICATION USAGE

8975 The *gethostent()* function may return pointers to static data, which may be overwritten by
8976 subsequent calls to any of these functions.

8977 RATIONALE

8978 None.

8979 FUTURE DIRECTIONS

8980 None.

8981 SEE ALSO

8982 *endservent()*, *gethostbyaddr()*, the Base Definitions volume of IEEE Std 1003.1-2001, <netdb.h>

8983 **CHANGE HISTORY**

8984 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

8985 **NAME**

8986 endnetent, getnetbyaddr, getnetbyname, getnetent, setnetent — network database functions

8987 **SYNOPSIS**

8988 #include <netdb.h>

8989 void endnetent(void);

8990 struct netent *getnetbyaddr(uint32_t net, int type);

8991 struct netent *getnetbyname(const char *name);

8992 struct netent *getnetent(void);

8993 void setnetent(int stayopen);

8994 **DESCRIPTION**

8995 These functions shall retrieve information about networks. This information is considered to be
 8996 stored in a database that can be accessed sequentially or randomly. The implementation of this
 8997 database is unspecified.

8998 The *setnetent()* function shall open and rewind the database. If the *stayopen* argument is non-
 8999 zero, the connection to the *net* database shall not be closed after each call to *getnetent()* (either
 9000 directly, or indirectly through one of the other *getnet**() functions), and the implementation may
 9001 maintain an open file descriptor to the database.

9002 The *getnetent()* function shall read the next entry of the database, opening and closing a
 9003 connection to the database as necessary.

9004 The *getnetbyaddr()* function shall search the database from the beginning, and find the first entry
 9005 for which the address family specified by *type* matches the *n_addrtype* member and the network
 9006 number *net* matches the *n_net* member, opening and closing a connection to the database as
 9007 necessary. The *net* argument shall be the network number in host byte order.

9008 The *getnetbyname()* function shall search the database from the beginning and find the first entry
 9009 for which the network name specified by *name* matches the *n_name* member, opening and
 9010 closing a connection to the database as necessary.

9011 The *getnetbyaddr()*, *getnetbyname()*, and *getnetent()* functions shall each return a pointer to a
 9012 **netent** structure, the members of which shall contain the fields of an entry in the network
 9013 database.

9014 The *endnetent()* function shall close the database, releasing any open file descriptor.

9015 These functions need not be reentrant. A function that is not required to be reentrant is not
 9016 required to be thread-safe.

9017 **RETURN VALUE**

9018 Upon successful completion, *getnetbyaddr()*, *getnetbyname()*, and *getnetent()* shall return a
 9019 pointer to a **netent** structure if the requested entry was found, and a null pointer if the end of the
 9020 database was reached or the requested entry was not found. Otherwise, a null pointer shall be
 9021 returned.

9022 **ERRORS**

9023 No errors are defined.

9024 **EXAMPLES**

9025 None.

9026 **APPLICATION USAGE**

9027 The *getnetbyaddr()*, *getnetbyname()*, and *getnetent()* functions may return pointers to static data,
9028 which may be overwritten by subsequent calls to any of these functions.

9029 **RATIONALE**

9030 None.

9031 **FUTURE DIRECTIONS**

9032 None.

9033 **SEE ALSO**9034 The Base Definitions volume of IEEE Std 1003.1-2001, <**netdb.h**>9035 **CHANGE HISTORY**

9036 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

9037 **NAME**

9038 endprotoent, getprotobyname, getprotobynumber, getprotoent, setprotoent — network protocol
9039 database functions

9040 **SYNOPSIS**

```
9041 #include <netdb.h>

9042 void endprotoent(void);
9043 struct protoent *getprotobyname(const char *name);
9044 struct protoent *getprotobynumber(int proto);
9045 struct protoent *getprotoent(void);
9046 void setprotoent(int stayopen);
```

9047 **DESCRIPTION**

9048 These functions shall retrieve information about protocols. This information is considered to be
9049 stored in a database that can be accessed sequentially or randomly. The implementation of this
9050 database is unspecified.

9051 The *setprotoent()* function shall open a connection to the database, and set the next entry to the
9052 first entry. If the *stayopen* argument is non-zero, the connection to the network protocol database
9053 shall not be closed after each call to *getprotoent()* (either directly, or indirectly through one of the
9054 other *getproto**() functions), and the implementation may maintain an open file descriptor for
9055 the database.

9056 The *getprotobyname()* function shall search the database from the beginning and find the first
9057 entry for which the protocol name specified by *name* matches the *p_name* member, opening and
9058 closing a connection to the database as necessary.

9059 The *getprotobynumber()* function shall search the database from the beginning and find the first
9060 entry for which the protocol number specified by *proto* matches the *p_proto* member, opening
9061 and closing a connection to the database as necessary.

9062 The *getprotoent()* function shall read the next entry of the database, opening and closing a
9063 connection to the database as necessary.

9064 The *getprotobyname()*, *getprotobynumber()*, and *getprotoent()* functions shall each return a pointer
9065 to a **protoent** structure, the members of which shall contain the fields of an entry in the network
9066 protocol database.

9067 The *endprotoent()* function shall close the connection to the database, releasing any open file
9068 descriptor.

9069 These functions need not be reentrant. A function that is not required to be reentrant is not
9070 required to be thread-safe.

9071 **RETURN VALUE**

9072 Upon successful completion, *getprotobyname()*, *getprotobynumber()*, and *getprotoent()* return a
9073 pointer to a **protoent** structure if the requested entry was found, and a null pointer if the end of
9074 the database was reached or the requested entry was not found. Otherwise, a null pointer is
9075 returned.

9076 **ERRORS**

9077 No errors are defined.

9078 EXAMPLES

9079 None.

9080 APPLICATION USAGE

9081 The *getprotobyname()*, *getprotobynumber()*, and *getprotoent()* functions may return pointers to
9082 static data, which may be overwritten by subsequent calls to any of these functions.

9083 RATIONALE

9084 None.

9085 FUTURE DIRECTIONS

9086 None.

9087 SEE ALSO

9088 The Base Definitions volume of IEEE Std 1003.1-2001, <**netdb.h**>

9089 CHANGE HISTORY

9090 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

9091 NAME

9092 endpwent, getpwent, setpwent — user database functions

9093 SYNOPSIS

9094 XSI `#include <pwd.h>`

```

9095 void endpwent(void);
9096 struct passwd *getpwent(void);
9097 void setpwent(void);
9098

```

9099 DESCRIPTION

9100 These functions shall retrieve information about users.

9101 The *getpwent()* function shall return a pointer to a structure containing the broken-out fields of
 9102 an entry in the user database. Each entry in the user database contains a **passwd** structure. When
 9103 first called, *getpwent()* shall return a pointer to a **passwd** structure containing the first entry in
 9104 the user database. Thereafter, it shall return a pointer to a **passwd** structure containing the next
 9105 entry in the user database. Successive calls can be used to search the entire user database.

9106 If an end-of-file or an error is encountered on reading, *getpwent()* shall return a null pointer.

9107 An implementation that provides extended security controls may impose further
 9108 implementation-defined restrictions on accessing the user database. In particular, the system
 9109 may deny the existence of some or all of the user database entries associated with users other
 9110 than the caller.

9111 The *setpwent()* function effectively rewinds the user database to allow repeated searches.9112 The *endpwent()* function may be called to close the user database when processing is complete.

9113 These functions need not be reentrant. A function that is not required to be reentrant is not
 9114 required to be thread-safe.

9115 RETURN VALUE

9116 The *getpwent()* function shall return a null pointer on end-of-file or error.

9117 ERRORS

9118 The *getpwent()*, *setpwent()*, and *endpwent()* functions may fail if:

9119 [EIO] An I/O error has occurred.

9120 In addition, *getpwent()* and *setpwent()* may fail if:

9121 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

9122 [ENFILE] The maximum allowable number of files is currently open in the system.

9123 The return value may point to a static area which is overwritten by a subsequent call to
 9124 *getpwuid()*, *getpwnam()*, or *getpwent()*.

9125 **EXAMPLES**9126 **Searching the User Database**

9127 The following example uses the *getpwent()* function to get successive entries in the user
9128 database, returning a pointer to a **passwd** structure that contains information about each user.
9129 The call to *endpwent()* closes the user database and cleans up.

```
9130 #include <pwd.h>
9131 ...
9132 struct passwd *p;
9133 ...
9134 while ((p = getpwent ()) != NULL) {
9135     ...
9136 }
9137 endpwent();
9138 ...
```

9139 **APPLICATION USAGE**

9140 These functions are provided due to their historical usage. Applications should avoid
9141 dependencies on fields in the password database, whether the database is a single file, or where
9142 in the file system name space the database resides. Applications should use *getpwuid()*
9143 whenever possible because it avoids these dependencies.

9144 **RATIONALE**

9145 None.

9146 **FUTURE DIRECTIONS**

9147 None.

9148 **SEE ALSO**

9149 *endgrent()*, *getlogin()*, *getpwnam()*, *getpwuid()*, the Base Definitions volume of
9150 IEEE Std 1003.1-2001, <pwd.h>

9151 **CHANGE HISTORY**

9152 First released in Issue 4, Version 2.

9153 **Issue 5**

9154 Moved from X/OPEN UNIX extension to BASE.

9155 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
9156 VALUE section.

9157 A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

9158 **Issue 6**

9159 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

9160 **NAME**

9161 endservent, getservbyname, getservbyport, getservent, setservent — network services database
9162 functions

9163 **SYNOPSIS**

```
9164 #include <netdb.h>

9165 void endservent(void);
9166 struct servent *getservbyname(const char *name, const char *proto);
9167 struct servent *getservbyport(int port, const char *proto);
9168 struct servent *getservent(void);
9169 void setservent(int stayopen);
```

9170 **DESCRIPTION**

9171 These functions shall retrieve information about network services. This information is
9172 considered to be stored in a database that can be accessed sequentially or randomly. The
9173 implementation of this database is unspecified.

9174 The *setservent()* function shall open a connection to the database, and set the next entry to the
9175 first entry. If the *stayopen* argument is non-zero, the *net* database shall not be closed after each
9176 call to the *getservent()* function (either directly, or indirectly through one of the other *getserv*()*
9177 functions), and the implementation may maintain an open file descriptor for the database.

9178 The *getservent()* function shall read the next entry of the database, opening and closing a
9179 connection to the database as necessary.

9180 The *getservbyname()* function shall search the database from the beginning and find the first
9181 entry for which the service name specified by *name* matches the *s_name* member and the protocol
9182 name specified by *proto* matches the *s_proto* member, opening and closing a connection to the
9183 database as necessary. If *proto* is a null pointer, any value of the *s_proto* member shall be
9184 matched.

9185 The *getservbyport()* function shall search the database from the beginning and find the first entry
9186 for which the port specified by *port* matches the *s_port* member and the protocol name specified
9187 by *proto* matches the *s_proto* member, opening and closing a connection to the database as
9188 necessary. If *proto* is a null pointer, any value of the *s_proto* member shall be matched. The *port*
9189 argument shall be in network byte order.

9190 The *getservbyname()*, *getservbyport()*, and *getservent()* functions shall each return a pointer to a
9191 **servent** structure, the members of which shall contain the fields of an entry in the network
9192 services database.

9193 The *endservent()* function shall close the database, releasing any open file descriptor.

9194 These functions need not be reentrant. A function that is not required to be reentrant is not
9195 required to be thread-safe.

9196 **RETURN VALUE**

9197 Upon successful completion, *getservbyname()*, *getservbyport()*, and *getservent()* return a pointer to
9198 a **servent** structure if the requested entry was found, and a null pointer if the end of the database
9199 was reached or the requested entry was not found. Otherwise, a null pointer is returned.

9200 **ERRORS**

9201 No errors are defined.

9202 **EXAMPLES**

9203 None.

9204 **APPLICATION USAGE**9205 The *port* argument of *getservbyport()* need not be compatible with the port values of all address
9206 families.9207 The *getservbyname()*, *getservbyport()*, and *getservent()* functions may return pointers to static
9208 data, which may be overwritten by subsequent calls to any of these functions.9209 **RATIONALE**

9210 None.

9211 **FUTURE DIRECTIONS**

9212 None.

9213 **SEE ALSO**9214 *endhostent()*, *endprotoent()*, *htonl()*, *inet_addr()*, the Base Definitions volume of
9215 IEEE Std 1003.1-2001, <netdb.h>9216 **CHANGE HISTORY**

9217 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

9218 NAME

9219 endutxent, getutxent, getutxid, getutxline, pututxline, setutxent — user accounting database
 9220 functions

9221 SYNOPSIS

```
9222 xSI      #include <utmpx.h>

9223      void endutxent(void);
9224      struct utmpx *getutxent(void);
9225      struct utmpx *getutxid(const struct utmpx *id);
9226      struct utmpx *getutxline(const struct utmpx *line);
9227      struct utmpx *pututxline(const struct utmpx *utmpx);
9228      void setutxent(void);
9229
```

9230 DESCRIPTION

9231 These functions shall provide access to the user accounting database.

9232 The *getutxent()* function shall read the next entry from the user accounting database. If the
 9233 database is not already open, it shall open it. If it reaches the end of the database, it shall fail.

9234 The *getutxid()* function shall search forward from the current point in the database. If the
 9235 *ut_type* value of the **utmpx** structure pointed to by *id* is *BOOT_TIME*, *OLD_TIME*, or
 9236 *NEW_TIME*, then it shall stop when it finds an entry with a matching *ut_type* value. If the
 9237 *ut_type* value is *INIT_PROCESS*, *LOGIN_PROCESS*, *USER_PROCESS*, or *DEAD_PROCESS*,
 9238 then it shall stop when it finds an entry whose type is one of these four and whose *ut_id* member
 9239 matches the *ut_id* member of the **utmpx** structure pointed to by *id*. If the end of the database is
 9240 reached without a match, *getutxid()* shall fail.

9241 The *getutxline()* function shall search forward from the current point in the database until it
 9242 finds an entry of the type *LOGIN_PROCESS* or *USER_PROCESS* which also has a *ut_line* value
 9243 matching that in the **utmpx** structure pointed to by *line*. If the end of the database is reached
 9244 without a match, *getutxline()* shall fail.

9245 The *getutxid()* or *getutxline()* function may cache data. For this reason, to use *getutxline()* to
 9246 search for multiple occurrences, the application shall zero out the static data after each success,
 9247 or *getutxline()* may return a pointer to the same **utmpx** structure.

9248 There is one exception to the rule about clearing the structure before further reads are done. The
 9249 implicit read done by *pututxline()* (if it finds that it is not already at the correct place in the user
 9250 accounting database) shall not modify the static structure returned by *getutxent()*, *getutxid()*, or
 9251 *getutxline()*, if the application has modified this structure and passed the pointer back to
 9252 *pututxline()*.

9253 For all entries that match a request, the *ut_type* member indicates the type of the entry. Other
 9254 members of the entry shall contain meaningful data based on the value of the *ut_type* member as
 9255 follows:

ut_type Member	Other Members with Meaningful Data
EMPTY	No others
BOOT_TIME	<i>ut_tv</i>
OLD_TIME	<i>ut_tv</i>
NEW_TIME	<i>ut_tv</i>
USER_PROCESS	<i>ut_id</i> , <i>ut_user</i> (login name of the user), <i>ut_line</i> , <i>ut_pid</i> , <i>ut_tv</i>
INIT_PROCESS	<i>ut_id</i> , <i>ut_pid</i> , <i>ut_tv</i>
LOGIN_PROCESS	<i>ut_id</i> , <i>ut_user</i> (implementation-defined name of the login process), <i>ut_pid</i> , <i>ut_tv</i>
DEAD_PROCESS	<i>ut_id</i> , <i>ut_pid</i> , <i>ut_tv</i>

An implementation that provides extended security controls may impose implementation-defined restrictions on accessing the user accounting database. In particular, the system may deny the existence of some or all of the user accounting database entries associated with users other than the caller.

If the process has appropriate privileges, the *pututxline()* function shall write out the structure into the user accounting database. It shall use *getutxid()* to search for a record that satisfies the request. If this search succeeds, then the entry shall be replaced. Otherwise, a new entry shall be made at the end of the user accounting database.

The *endutxent()* function shall close the user accounting database.

The *setutxent()* function shall reset the input to the beginning of the database. This should be done before each search for a new entry if it is desired that the entire database be examined.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, *getutxent()*, *getutxid()*, and *getutxline()* shall return a pointer to a **utmpx** structure containing a copy of the requested entry in the user accounting database. Otherwise, a null pointer shall be returned.

The return value may point to a static area which is overwritten by a subsequent call to *getutxid()* or *getutxline()*.

Upon successful completion, *pututxline()* shall return a pointer to a **utmpx** structure containing a copy of the entry added to the user accounting database. Otherwise, a null pointer shall be returned.

The *endutxent()* and *setutxent()* functions shall not return a value.

ERRORS

No errors are defined for the *endutxent()*, *getutxent()*, *getutxid()*, *getutxline()*, and *setutxent()* functions.

The *pututxline()* function may fail if:

[EPERM] The process does not have appropriate privileges.

9295 **EXAMPLES**

9296 None.

9297 **APPLICATION USAGE**9298 The sizes of the arrays in the structure can be found using the *sizeof* operator.9299 **RATIONALE**

9300 None.

9301 **FUTURE DIRECTIONS**

9302 None.

9303 **SEE ALSO**

9304 The Base Definitions volume of IEEE Std 1003.1-2001, <utmpx.h>

9305 **CHANGE HISTORY**

9306 First released in Issue 4, Version 2.

9307 **Issue 5**

9308 Moved from X/OPEN UNIX extension to BASE.

9309 Normative text previously in the APPLICATION USAGE section is moved to the
9310 DESCRIPTION.

9311 A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

9312 **Issue 6**

9313 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

9314 **NAME**

9315 environ — array of character pointers to the environment strings

9316 **SYNOPSIS**

9317 extern char **environ;

9318 **DESCRIPTION**9319 Refer to the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 8, Environment Variables
9320 and *exec*.

9321 **NAME**9322 **erand48** — generate uniformly distributed pseudo-random numbers9323 **SYNOPSIS**9324 xSI `#include <stdlib.h>`9325 `double erand48(unsigned short xsubi[3]);`

9326

9327 **DESCRIPTION**9328 Refer to *drand48()*.

9329 **NAME**

9330 erf, erff, erfl — error functions

9331 **SYNOPSIS**

9332 #include <math.h>

9333 double erf(double x);

9334 float erff(float x);

9335 long double erfl(long double x);

9336 **DESCRIPTION**

9337 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 9338 conflict between the requirements described here and the ISO C standard is unintentional. This
 9339 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

9340 These functions shall compute the error function of their argument *x*, defined as:

$$9341 \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

9342 An application wishing to check for error situations should set *errno* to zero and call
 9343 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 9344 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 9345 zero, an error has occurred.

9346 **RETURN VALUE**

9347 Upon successful completion, these functions shall return the value of the error function.

9348 **MX** If *x* is NaN, a NaN shall be returned.

9349 If *x* is ±0, ±0 shall be returned.

9350 If *x* is ±Inf, ±1 shall be returned.

9351 If *x* is subnormal, a range error may occur, and $2 * x / \sqrt{\pi}$ should be returned.

9352 **ERRORS**

9353 These functions may fail if:

9354 **MX** **Range Error** The result underflows.

9355 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 9356 then *errno* shall be set to [ERANGE]. If the integer expression
 9357 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
 9358 floating-point exception shall be raised.

9359 **EXAMPLES**

9360 None.

9361 **APPLICATION USAGE**

9362 Underflow occurs when $|x| < \text{DBL_MIN} * (\sqrt{\pi}/2)$.

9363 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
 9364 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

9365 **RATIONALE**

9366 None.

9367 **FUTURE DIRECTIONS**

9368 None.

9369 **SEE ALSO**

9370 *erfc()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-2001,
9371 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

9372 **CHANGE HISTORY**

9373 First released in Issue 1. Derived from Issue 1 of the SVID.

9374 **Issue 5**

9375 The DESCRIPTION is updated to indicate how an application should check for an error. This
9376 text was previously published in the APPLICATION USAGE section.

9377 **Issue 6**9378 The *erf()* function is no longer marked as an extension.9379 The *erfc()* function is now split out onto its own reference page.9380 The *erff()* and *erfl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

9381 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
9382 revised to align with the ISO/IEC 9899:1999 standard.

9383 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
9384 marked.

9385 **NAME**

9386 erfc, erfcf, erfc1 — complementary error functions

9387 **SYNOPSIS**

9388 #include <math.h>

9389 double erfc(double x);

9390 float erfcf(float x);

9391 long double erfc1(long double x);

9392 **DESCRIPTION**

9393 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 9394 conflict between the requirements described here and the ISO C standard is unintentional. This
 9395 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

9396 These functions shall compute the complementary error function $1.0 - \text{erf}(x)$.

9397 An application wishing to check for error situations should set *errno* to zero and call
 9398 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 9399 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 9400 zero, an error has occurred.

9401 **RETURN VALUE**9402 Upon successful completion, these functions shall return the value of the complementary error
9403 function.

9404 If the correct value would cause underflow and is not representable, a range error may occur
 9405 **MX** and either 0.0 (if representable), or an implementation-defined value shall be returned.

9406 **MX** If *x* is NaN, a NaN shall be returned.9407 If *x* is ± 0 , +1 shall be returned.9408 If *x* is $-\text{Inf}$, +2 shall be returned.9409 If *x* is $+\text{Inf}$, +0 shall be returned.

9410 If the correct value would cause underflow and is representable, a range error may occur and the
 9411 correct value shall be returned.

9412 **ERRORS**

9413 These functions may fail if:

9414 Range Error The result underflows.

9415 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 9416 then *errno* shall be set to [ERANGE]. If the integer expression
 9417 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
 9418 floating-point exception shall be raised.

9419 **EXAMPLES**

9420 None.

9421 **APPLICATION USAGE**

9422 The *erfc()* function is provided because of the extreme loss of relative accuracy if *erf(x)* is called
 9423 for large *x* and the result subtracted from 1.0.

9424 Note for IEEE Std 754-1985 **double**, $26.55 < x$ implies *erfc(x)* has underflowed.

9425 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
 9426 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

9427 **RATIONALE**

9428 None.

9429 **FUTURE DIRECTIONS**

9430 None.

9431 **SEE ALSO**9432 *erf()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-2001,
9433 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>9434 **CHANGE HISTORY**

9435 First released in Issue 1. Derived from Issue 1 of the SVID.

9436 **Issue 5**9437 The DESCRIPTION is updated to indicate how an application should check for an error. This
9438 text was previously published in the APPLICATION USAGE section.9439 **Issue 6**9440 The *erfc()* function is no longer marked as an extension.9441 These functions are split out from the *erf()* reference page.9442 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
9443 revised to align with the ISO/IEC 9899:1999 standard.9444 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
9445 marked.

9446 **NAME**

9447 erff, erfl — error functions

9448 **SYNOPSIS**

9449 #include <math.h>

9450 float erff(float *x*);9451 long double erfl(long double *x*);9452 **DESCRIPTION**9453 Refer to *erf()*.

9454 **NAME**9455 `errno` — error return value9456 **SYNOPSIS**9457 `#include <errno.h>`9458 **DESCRIPTION**9459 The lvalue *errno* is used by many functions to return error values.

9460 Many functions provide an error number in *errno*, which has type **int** and is defined in
9461 `<errno.h>`. The value of *errno* shall be defined only after a call to a function for which it is
9462 explicitly stated to be set and until it is changed by the next function call or if the application
9463 assigns it a value. The value of *errno* should only be examined when it is indicated to be valid by
9464 a function's return value. Applications shall obtain the definition of *errno* by the inclusion of
9465 `<errno.h>`. No function in this volume of IEEE Std 1003.1-2001 shall set *errno* to 0.

9466 It is unspecified whether *errno* is a macro or an identifier declared with external linkage. If a
9467 macro definition is suppressed in order to access an actual object, or a program defines an
9468 identifier with the name *errno*, the behavior is undefined.

9469 The symbolic values stored in *errno* are documented in the ERRORS sections on all relevant
9470 pages.

9471 **RETURN VALUE**

9472 None.

9473 **ERRORS**

9474 None.

9475 **EXAMPLES**

9476 None.

9477 **APPLICATION USAGE**

9478 Previously both POSIX and X/Open documents were more restrictive than the ISO C standard
9479 in that they required *errno* to be defined as an external variable, whereas the ISO C standard
9480 required only that *errno* be defined as a modifiable lvalue with type **int**.

9481 An application that needs to examine the value of *errno* to determine the error should set it to 0
9482 before a function call, then inspect it before a subsequent function call.

9483 **RATIONALE**

9484 None.

9485 **FUTURE DIRECTIONS**

9486 None.

9487 **SEE ALSO**9488 Section 2.3, the Base Definitions volume of IEEE Std 1003.1-2001, `<errno.h>`9489 **CHANGE HISTORY**

9490 First released in Issue 1. Derived from Issue 1 of the SVID.

9491 **Issue 5**

9492 The following sentence is deleted from the DESCRIPTION: “The value of *errno* is 0 at program
9493 start-up, but is never set to 0 by any XSI function”. The DESCRIPTION also no longer states that
9494 conforming implementations may support the declaration:

9495 `extern int errno;`

Issue 6

Obsolescent text regarding defining *errno* as:

```
extern int errno
```

is removed.

Text regarding no function setting *errno* to zero to indicate an error is changed to no function shall set *errno* to zero. This is for alignment with the ISO/IEC 9899:1999 standard.

9502 NAME

9503 environ, execl, execv, execl, execve, execlp, execvp — execute a file

9504 SYNOPSIS

```

9505     #include <unistd.h>

9506     extern char **environ;
9507     int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
9508     int execv(const char *path, char *const argv[]);
9509     int execl(const char *path, const char *arg0, ... /*,
9510               (char *)0, char *const envp[] */);
9511     int execve(const char *path, char *const argv[], char *const envp[]);
9512     int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
9513     int execvp(const char *file, char *const argv[]);

```

9514 DESCRIPTION

9515 The *exec* family of functions shall replace the current process image with a new process image.
 9516 The new image shall be constructed from a regular, executable file called the *new process image*
 9517 *file*. There shall be no return from a successful *exec*, because the calling process image is overlaid
 9518 by the new process image.

9519 When a C-language program is executed as a result of this call, it shall be entered as a C-
 9520 language function call as follows:

```

9521     int main (int argc, char *argv[]);

```

9522 where *argc* is the argument count and *argv* is an array of character pointers to the arguments
 9523 themselves. In addition, the following variable:

```

9524     extern char **environ;

```

9525 is initialized as a pointer to an array of character pointers to the environment strings. The *argv*
 9526 and *environ* arrays are each terminated by a null pointer. The null pointer terminating the *argv*
 9527 array is not counted in *argc*.

9528 THR Conforming multi-threaded applications shall not use the *environ* variable to access or modify
 9529 any environment variable while any other thread is concurrently modifying any environment
 9530 variable. A call to any function dependent on any environment variable shall be considered a use
 9531 of the *environ* variable to access that environment variable.

9532 The arguments specified by a program with one of the *exec* functions shall be passed on to the
 9533 new process image in the corresponding *main()* arguments.

9534 The argument *path* points to a pathname that identifies the new process image file.

9535 The argument *file* is used to construct a pathname that identifies the new process image file. If
 9536 the *file* argument contains a slash character, the *file* argument shall be used as the pathname for
 9537 this file. Otherwise, the path prefix for this file is obtained by a search of the directories passed
 9538 as the environment variable *PATH* (see the Base Definitions volume of IEEE Std 1003.1-2001,
 9539 Chapter 8, Environment Variables). If this environment variable is not present, the results of the
 9540 search are implementation-defined.

9541 There are two distinct ways in which the contents of the process image file may cause the
 9542 execution to fail, distinguished by the setting of *errno* to either [ENOEXEC] or [EINVAL] (see the
 9543 ERRORS section). In the cases where the other members of the *exec* family of functions would
 9544 fail and set *errno* to [ENOEXEC], the *execlp()* and *execvp()* functions shall execute a command
 9545 interpreter and the environment of the executed command shall be as if the process invoked the
 9546 *sh* utility using *execl()* as follows:

9547 `execl(<shell path>, arg0, file, arg1, ..., (char *)0);`

9548 where *<shell path>* is an unspecified pathname for the *sh* utility, *file* is the process image file, and
 9549 for *execvp()*, where *arg0*, *arg1*, and so on correspond to the values passed to *execvp()* in *argv*[0],
 9550 *argv*[1], and so on.

9551 The arguments represented by *arg0*,... are pointers to null-terminated character strings. These
 9552 strings shall constitute the argument list available to the new process image. The list is
 9553 terminated by a null pointer. The argument *arg0* should point to a filename that is associated
 9554 with the process being started by one of the *exec* functions.

9555 The argument *argv* is an array of character pointers to null-terminated strings. The application
 9556 shall ensure that the last member of this array is a null pointer. These strings shall constitute the
 9557 argument list available to the new process image. The value in *argv*[0] should point to a filename
 9558 that is associated with the process being started by one of the *exec* functions.

9559 The argument *envp* is an array of character pointers to null-terminated strings. These strings
 9560 shall constitute the environment for the new process image. The *envp* array is terminated by a
 9561 null pointer.

9562 For those forms not containing an *envp* pointer (*execl()*, *execv()*, *execlp()*, and *execvp()*), the
 9563 environment for the new process image shall be taken from the external variable *environ* in the
 9564 calling process.

9565 The number of bytes available for the new process' combined argument and environment lists is
 9566 {ARG_MAX}. It is implementation-defined whether null terminators, pointers, and/or any
 9567 alignment bytes are included in this total.

9568 File descriptors open in the calling process image shall remain open in the new process image,
 9569 except for those whose close-on-exec flag FD_CLOEXEC is set. For those file descriptors that
 9570 remain open, all attributes of the open file description remain unchanged. For any file descriptor
 9571 that is closed for this reason, file locks are removed as a result of the close as described in *close()*.
 9572 Locks that are not removed by closing of file descriptors remain unchanged.

9573 Directory streams open in the calling process image shall be closed in the new process image.

9574 The state of the floating-point environment in the new process image shall be set to the default.

9575 XSI The state of conversion descriptors and message catalog descriptors in the new process image is
 9576 undefined. For the new process image, the equivalent of:

9577 `setlocale(LC_ALL, "C")`

9578 shall be executed at start-up.

9579 Signals set to the default action (SIG_DFL) in the calling process image shall be set to the default
 9580 action in the new process image. Except for SIGCHLD, signals set to be ignored (SIG_IGN) by
 9581 the calling process image shall be set to be ignored by the new process image. Signals set to be
 9582 caught by the calling process image shall be set to the default action in the new process image
 9583 (see <signal.h>). If the SIGCHLD signal is set to be ignored by the calling process image, it is
 9584 unspecified whether the SIGCHLD signal is set to be ignored or to the default action in the new
 9585 XSI process image. After a successful call to any of the *exec* functions, alternate signal stacks are not
 9586 preserved and the SA_ONSTACK flag shall be cleared for all signals.

9587 After a successful call to any of the *exec* functions, any functions previously registered by *atexit()*
 9588 are no longer registered.

9589 XSI If the ST_NOSUID bit is set for the file system containing the new process image file, then the
 9590 effective user ID, effective group ID, saved set-user-ID, and saved set-group-ID are unchanged
 9591 in the new process image. Otherwise, if the set-user-ID mode bit of the new process image file is

9592		set, the effective user ID of the new process image shall be set to the user ID of the new process image file. Similarly, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process image shall be set to the group ID of the new process image file. The real user ID, real group ID, and supplementary group IDs of the new process image shall remain the same as those of the calling process image. The effective user ID and effective group ID of the new process image shall be saved (as the saved set-user-ID and the saved set-group-ID) for use by <i>setuid()</i> .
9599	XSI	Any shared memory segments attached to the calling process image shall not be attached to the new process image.
9600		
9601	SEM	Any named semaphores open in the calling process shall be closed as if by appropriate calls to <i>sem_close()</i> .
9602		
9603	TYM	Any blocks of typed memory that were mapped in the calling process are unmapped, as if <i>munmap()</i> was implicitly called to unmap them.
9604		
9605	ML	Memory locks established by the calling process via calls to <i>mlockall()</i> or <i>mlock()</i> shall be removed. If locked pages in the address space of the calling process are also mapped into the address spaces of other processes and are locked by those processes, the locks established by the other processes shall be unaffected by the call by this process to the <i>exec</i> function. If the <i>exec</i> function fails, the effect on memory locks is unspecified.
9606		
9607		
9608		
9609		
9610	MF SHM	Memory mappings created in the process are unmapped before the address space is rebuilt for the new process image.
9611		
9612	PS	For the SCHED_FIFO and SCHED_RR scheduling policies, the policy and priority settings shall not be changed by a call to an <i>exec</i> function. For other scheduling policies, the policy and priority settings on <i>exec</i> are implementation-defined.
9613		
9614		
9615	TMR	Per-process timers created by the calling process shall be deleted before replacing the current process image with the new process image.
9616		
9617	MSG	All open message queue descriptors in the calling process shall be closed, as described in <i>mq_close()</i> .
9618		
9619	AIO	Any outstanding asynchronous I/O operations may be canceled. Those asynchronous I/O operations that are not canceled shall complete as if the <i>exec</i> function had not yet occurred, but any associated signal notifications shall be suppressed. It is unspecified whether the <i>exec</i> function itself blocks awaiting such I/O completion. In no event, however, shall the new process image created by the <i>exec</i> function be affected by the presence of outstanding asynchronous I/O operations at the time the <i>exec</i> function is called. Whether any I/O is canceled, and which I/O may be canceled upon <i>exec</i> , is implementation-defined.
9620		
9621		
9622		
9623		
9624		
9625		
9626	CPT	The new process image shall inherit the CPU-time clock of the calling process image. This inheritance means that the process CPU-time clock of the process being <i>exec</i> -ed shall not be reinitialized or altered as a result of the <i>exec</i> function other than to reflect the time spent by the process executing the <i>exec</i> function itself.
9627		
9628		
9629		
9630	TCT	The initial value of the CPU-time clock of the initial thread of the new process image shall be set to zero.
9631		
9632	TRC	If the calling process is being traced, the new process image shall continue to be traced into the same trace stream as the original process image, but the new process image shall not inherit the mapping of trace event names to trace event type identifiers that was defined by calls to the <i>posix_trace_eventid_open()</i> or the <i>posix_trace_trid_eventid_open()</i> functions in the calling process image.
9633		
9634		
9635		
9636		

9637 If the calling process is a trace controller process, any trace streams that were created by the
 9638 calling process shall be shut down as described in the *posix_trace_shutdown()* function.

9639 The new process shall inherit at least the following attributes from the calling process image:

- 9640 XSI • Nice value (see *nice()*)
- 9641 XSI • *semadj* values (see *semop()*)
- 9642 • Process ID
- 9643 • Parent process ID
- 9644 • Process group ID
- 9645 • Session membership
- 9646 • Real user ID
- 9647 • Real group ID
- 9648 • Supplementary group IDs
- 9649 • Time left until an alarm clock signal (see *alarm()*)
- 9650 • Current working directory
- 9651 • Root directory
- 9652 • File mode creation mask (see *umask()*)
- 9653 XSI • File size limit (see *ulimit()*)
- 9654 • Process signal mask (see *sigprocmask()*)
- 9655 • Pending signal (see *sigpending()*)
- 9656 • *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* (see *times()*)
- 9657 XSI • Resource limits
- 9658 XSI • Controlling terminal
- 9659 XSI • Interval timers

9660 All other process attributes defined in this volume of IEEE Std 1003.1-2001 shall be the same in
 9661 the new and old process images. The inheritance of process attributes not defined by this
 9662 volume of IEEE Std 1003.1-2001 is implementation-defined.

9663 A call to any *exec* function from a process with more than one thread shall result in all threads
 9664 being terminated and the new executable image being loaded and executed. No destructor
 9665 functions shall be called.

9666 Upon successful completion, the *exec* functions shall mark for update the *st_atime* field of the file.
 9667 If an *exec* function failed but was able to locate the process image file, whether the *st_atime* field
 9668 is marked for update is unspecified. Should the *exec* function succeed, the process image file
 9669 shall be considered to have been opened with *open()*. The corresponding *close()* shall be
 9670 considered to occur at a time after this open, but before process termination or successful
 9671 completion of a subsequent call to one of the *exec* functions, *posix_spawn()*, or *posix_spawnp()*.
 9672 The *argv[]* and *envp[]* arrays of pointers and the strings to which those arrays point shall not be
 9673 modified by a call to one of the *exec* functions, except as a consequence of replacing the process
 9674 image.

9675 XSI The saved resource limits in the new process image are set to be a copy of the process'
 9676 corresponding hard and soft limits.

9677 **RETURN VALUE**

9678 If one of the *exec* functions returns to the calling process image, an error has occurred; the return
 9679 value shall be `-1`, and *errno* shall be set to indicate the error.

9680 **ERRORS**

9681 The *exec* functions shall fail if:

9682 [E2BIG] The number of bytes used by the new process image's argument list and
 9683 environment list is greater than the system-imposed limit of `{ARG_MAX}`
 9684 bytes.

9685 [EACCES] Search permission is denied for a directory listed in the new process image
 9686 file's path prefix, or the new process image file denies execution permission,
 9687 or the new process image file is not a regular file and the implementation does
 9688 not support execution of files of its type.

9689 [EINVAL] The new process image file has the appropriate permission and has a
 9690 recognized executable binary format, but the system does not support
 9691 execution of a file with this format.

9692 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path* or *file*
 9693 argument.

9694 [ENAMETOOLONG]
 9695 The length of the *path* or *file* arguments exceeds `{PATH_MAX}` or a pathname
 9696 component is longer than `{NAME_MAX}`.

9697 [ENOENT] A component of *path* or *file* does not name an existing file or *path* or *file* is an
 9698 empty string.

9699 [ENOTDIR] A component of the new process image file's path prefix is not a directory.

9700 The *exec* functions, except for *execlp()* and *execvp()*, shall fail if:

9701 [ENOEXEC] The new process image file has the appropriate access permission but has an
 9702 unrecognized format.

9703 The *exec* functions may fail if:

9704 [ELOOP] More than `{SYMLOOP_MAX}` symbolic links were encountered during
 9705 resolution of the *path* or *file* argument.

9706 [ENAMETOOLONG]
 9707 As a result of encountering a symbolic link in resolution of the *path* argument,
 9708 the length of the substituted pathname string exceeded `{PATH_MAX}`.

9709 [ENOMEM] The new process image requires more memory than is allowed by the
 9710 hardware or system-imposed memory management constraints.

9711 [ETXTBSY] The new process image file is a pure procedure (shared text) file that is
 9712 currently open for writing by some process.

9713 **EXAMPLES**9714 **Using execl()**

9715 The following example executes the *ls* command, specifying the pathname of the executable
9716 (*/bin/ls*) and using arguments supplied directly to the command to produce single-column
9717 output.

```
9718 #include <unistd.h>
9719 int ret;
9720 ...
9721 ret = execl ("/bin/ls", "ls", "-l", (char *)0);
```

9722 **Using execl()**

9723 The following example is similar to **Using execl()**. In addition, it specifies the environment for
9724 the new process image using the *env* argument.

```
9725 #include <unistd.h>
9726 int ret;
9727 char *env[] = { "HOME=/usr/home", "LOGNAME=home", (char *)0 };
9728 ...
9729 ret = execl ("/bin/ls", "ls", "-l", (char *)0, env);
```

9730 **Using execlp()**

9731 The following example searches for the location of the *ls* command among the directories
9732 specified by the *PATH* environment variable.

```
9733 #include <unistd.h>
9734 int ret;
9735 ...
9736 ret = execlp ("ls", "ls", "-l", (char *)0);
```

9737 **Using execv()**

9738 The following example passes arguments to the *ls* command in the *cmd* array.

```
9739 #include <unistd.h>
9740 int ret;
9741 char *cmd[] = { "ls", "-l", (char *)0 };
9742 ...
9743 ret = execv ("/bin/ls", cmd);
```


Using `execve()`

The following example passes arguments to the *ls* command in the *cmd* array, and specifies the environment for the new process image using the *env* argument.

```
#include <unistd.h>

int ret;
char *cmd[] = { "ls", "-l", (char *)0 };
char *env[] = { "HOME=/usr/home", "LOGNAME=home", (char *)0 };
...
ret = execve ("/bin/ls", cmd, env);
```

Using `execvp()`

The following example searches for the location of the *ls* command among the directories specified by the *PATH* environment variable, and passes arguments to the *ls* command in the *cmd* array.

```
#include <unistd.h>

int ret;
char *cmd[] = { "ls", "-l", (char *)0 };
...
ret = execvp ("ls", cmd);
```

APPLICATION USAGE

As the state of conversion descriptors and message catalog descriptors in the new process image is undefined, conforming applications should not rely on their use and should close them prior to calling one of the *exec* functions.

Applications that require other than the default POSIX locale should call *setlocale()* with the appropriate parameters to establish the locale of the new process.

The *environ* array should not be accessed directly by the application.

RATIONALE

Early proposals required that the value of *argc* passed to *main()* be “one or greater”. This was driven by the same requirement in drafts of the ISO C standard. In fact, historical implementations have passed a value of zero when no arguments are supplied to the caller of the *exec* functions. This requirement was removed from the ISO C standard and subsequently removed from this volume of IEEE Std 1003.1-2001 as well. The wording, in particular the use of the word *should*, requires a Strictly Conforming POSIX Application to pass at least one argument to the *exec* function, thus guaranteeing that *argc* be one or greater when invoked by such an application. In fact, this is good practice, since many existing applications reference *argv[0]* without first checking the value of *argc*.

The requirement on a Strictly Conforming POSIX Application also states that the value passed as the first argument be a filename associated with the process being started. Although some existing applications pass a pathname rather than a filename in some circumstances, a filename is more generally useful, since the common usage of *argv[0]* is in printing diagnostics. In some cases the filename passed is not the actual filename of the file; for example, many implementations of the *login* utility use a convention of prefixing a hyphen (‘-’) to the actual filename, which indicates to the command interpreter being invoked that it is a “login shell”.

Historically there have been two ways that implementations can *exec* shell scripts.

One common historical implementation is that the *execl()*, *execv()*, *execle()*, and *execve()* functions return an [ENOEXEC] error for any file not recognizable as executable, including a shell script. When the *execlp()* and *execvp()* functions encounter such a file, they assume the file to be a shell script and invoke a known command interpreter to interpret such files. This is now required by IEEE Std 1003.1-2001. These implementations of *execvp()* and *execlp()* only give the [ENOEXEC] error in the rare case of a problem with the command interpreter's executable file. Because of these implementations, the [ENOEXEC] error is not mentioned for *execlp()* or *execvp()*, although implementations can still give it.

Another way that some historical implementations handle shell scripts is by recognizing the first two bytes of the file as the character string "#!" and using the remainder of the first line of the file as the name of the command interpreter to execute.

One potential source of confusion noted by the standard developers is over how the contents of a process image file affect the behavior of the *exec* family of functions. The following is a description of the actions taken:

1. If the process image file is a valid executable (in a format that is executable and valid and having appropriate permission) for this system, then the system executes the file.
2. If the process image file has appropriate permission and is in a format that is executable but not valid for this system (such as a recognized binary for another architecture), then this is an error and *errno* is set to [EINVAL] (see later RATIONALE on [EINVAL]).
3. If the process image file has appropriate permission but is not otherwise recognized:
 - a. If this is a call to *execlp()* or *execvp()*, then they invoke a command interpreter assuming that the process image file is a shell script.
 - b. If this is not a call to *execlp()* or *execvp()*, then an error occurs and *errno* is set to [ENOEXEC].

Applications that do not require to access their arguments may use the form:

```
main(void)
```

as specified in the ISO C standard. However, the implementation will always provide the two arguments *argc* and *argv*, even if they are not used.

Some implementations provide a third argument to *main()* called *envp*. This is defined as a pointer to the environment. The ISO C standard specifies invoking *main()* with two arguments, so implementations must support applications written this way. Since this volume of IEEE Std 1003.1-2001 defines the global variable *environ*, which is also provided by historical implementations and can be used anywhere that *envp* could be used, there is no functional need for the *envp* argument. Applications should use the *getenv()* function rather than accessing the environment directly via either *envp* or *environ*. Implementations are required to support the two-argument calling sequence, but this does not prohibit an implementation from supporting *envp* as an optional third argument.

This volume of IEEE Std 1003.1-2001 specifies that signals set to SIG_IGN remain set to SIG_IGN, and that the process signal mask be unchanged across an *exec*. This is consistent with historical implementations, and it permits some useful functionality, such as the *nohup* command. However, it should be noted that many existing applications wrongly assume that they start with certain signals set to the default action and/or unblocked. In particular, applications written with a simpler signal model that does not include blocking of signals, such as the one in the ISO C standard, may not behave properly if invoked with some signals blocked. Therefore, it is best not to block or ignore signals across *execs* without explicit reason to do so, and especially not to block signals across *execs* of arbitrary (not closely co-operating) programs.

The *exec* functions always save the value of the effective user ID and effective group ID of the process at the completion of the *exec*, whether or not the set-user-ID or the set-group-ID bit of the process image file is set.

The statement about *argv[]* and *envp[]* being constants is included to make explicit to future writers of language bindings that these objects are completely constant. Due to a limitation of the ISO C standard, it is not possible to state that idea in standard C. Specifying two levels of *const-qualification* for the *argv[]* and *envp[]* parameters for the *exec* functions may seem to be the natural choice, given that these functions do not modify either the array of pointers or the characters to which the function points, but this would disallow existing correct code. Instead, only the array of pointers is noted as constant. The table of assignment compatibility for *dst=src* derived from the ISO C standard summarizes the compatibility:

<i>dst:</i>	char *[]	const char *[]	char *const[]	const char *const[]
<i>src:</i>				
char *[]	VALID	—	VALID	—
const char *[]	—	VALID	—	VALID
char * const []	—	—	VALID	—
const char *const[]	—	—	—	VALID

Since all existing code has a source type matching the first row, the column that gives the most valid combinations is the third column. The only other possibility is the fourth column, but using it would require a cast on the *argv* or *envp* arguments. It is unfortunate that the fourth column cannot be used, because the declaration a non-expert would naturally use would be that in the second row.

The ISO C standard and this volume of IEEE Std 1003.1-2001 do not conflict on the use of *environ*, but some historical implementations of *environ* may cause a conflict. As long as *environ* is treated in the same way as an entry point (for example, *fork()*), it conforms to both standards. A library can contain *fork()*, but if there is a user-provided *fork()*, that *fork()* is given precedence and no problem ensues. The situation is similar for *environ*: the definition in this volume of IEEE Std 1003.1-2001 is to be used if there is no user-provided *environ* to take precedence. At least three implementations are known to exist that solve this problem.

[E2BIG] The limit {ARG_MAX} applies not just to the size of the argument list, but to the sum of that and the size of the environment list.

[EFAULT] Some historical systems return [EFAULT] rather than [ENOEXEC] when the new process image file is corrupted. They are non-conforming.

[EINVAL] This error condition was added to IEEE Std 1003.1-2001 to allow an implementation to detect executable files generated for different architectures, and indicate this situation to the application. Historical implementations of shells, *execvp()*, and *execlp()* that encounter an [ENOEXEC] error will execute a shell on the assumption that the file is a shell script. This will not produce the desired effect when the file is a valid executable for a different architecture. An implementation may now choose to avoid this problem by returning [EINVAL] when a valid executable for a different architecture is encountered. Some historical implementations return [EINVAL] to indicate that the *path* argument contains a character with the high order bit set. The standard developers chose to deviate from historical practice for the following reasons:

1. The new utilization of [EINVAL] will provide some measure of utility to the user community.

2. Historical use of [EINVAL] is not acceptable in an internationalized operating environment.

[ENAMETOOLONG]

Since the file pathname may be constructed by taking elements in the *PATH* variable and putting them together with the filename, the [ENAMETOOLONG] error condition could also be reached this way.

[ETXTBSY]

System V returns this error when the executable file is currently open for writing by some process. This volume of IEEE Std 1003.1-2001 neither requires nor prohibits this behavior.

Other systems (such as System V) may return [EINTR] from *exec*. This is not addressed by this volume of IEEE Std 1003.1-2001, but implementations may have a window between the call to *exec* and the time that a signal could cause one of the *exec* calls to return with [EINTR].

An explicit statement regarding the floating-point environment (as defined in the *<fenv.h>* header) was added to make it clear that the floating-point environment is set to its default when a call to one of the *exec* functions succeeds. The requirements for inheritance or setting to the default for other process and thread start-up functions is covered by more generic statements in their descriptions and can be summarized as follows:

posix_spawn() Set to default.

fork() Inherit.

pthread_create() Inherit.

FUTURE DIRECTIONS

None.

SEE ALSO

alarm(), *atexit()*, *chmod()*, *close()*, *exit()*, *fcntl()*, *fork()*, *fstatvfs()*, *getenv()*, *getitimer()*, *getrlimit()*, *mmap()*, *nice()*, *posix_spawn()*, *posix_trace_eventid_open()*, *posix_trace_shutdown()*, *posix_trace_trid_eventid_open()*, *putenv()*, *semop()*, *setlocale()*, *shmat()*, *sigaction()*, *sigaltstack()*, *sigpending()*, *sigprocmask()*, *system()*, *times()*, *ulimit()*, *umask()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface, *<unistd.h>*

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Large File Summit extensions are added.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, behavior is defined for when the process image file is not a valid executable.
- In this issue, *_POSIX_SAVED_IDS* is mandated, thus the effective user ID and effective group ID of the new process image shall be saved (as the saved set-user-ID and the saved set-group-ID) for use by the *setuid()* function.
- The [ELOOP] mandatory error condition is added.

- 9923
 - A second [ENAMETOOLONG] is added as an optional error condition.
- 9924
 - The [ETXTBSY] optional error condition is added.
- 9925

The following changes were made to align with the IEEE P1003.1a draft standard:
- 9926
 - The [EINVAL] mandatory error condition is added.
- 9927
 - The [ELOOP] optional error condition is added.
- 9928

The description of CPU-time clock semantics is added for alignment with IEEE Std 1003.1d-1999.
- 9929

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding semantics for
- 9930

typed memory.
- 9931

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
- 9932

The description of tracing semantics is added for alignment with IEEE Std 1003.1q-2000.
- 9933

IEEE PASC Interpretation 1003.1 #132 is applied.
- 9934

The DESCRIPTION is updated to make it explicit that the floating-point environment in the new
- 9935

process image is set to the default.
- 9936

The DESCRIPTION and RATIONALE are updated to include clarifications of how the contents
- 9937

of a process image file affect the behavior of the *exec* functions.

9938 **NAME**9939 `exit`, `_Exit`, `_exit` — terminate a process9940 **SYNOPSIS**9941 `#include <stdlib.h>`9942 `void exit(int status);`9943 `void _Exit(int status);`9944 `#include <unistd.h>`9945 `void _exit(int status);`9946 **DESCRIPTION**

9947 CX For `exit()` and `_Exit()`: The functionality described on this reference page is aligned with the
 9948 ISO C standard. Any conflict between the requirements described here and the ISO C standard is
 9949 unintentional. This volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

9950 CX The value of *status* may be 0, `EXIT_SUCCESS`, `EXIT_FAILURE`, or any other value, though only
 9951 the least significant 8 bits (that is, *status* & 0377) shall be available to a waiting parent process.

9952 The `exit()` function shall first call all functions registered by `atexit()`, in the reverse order of their
 9953 registration, except that a function is called after any previously registered functions that had
 9954 already been called at the time it was registered. Each function is called as many times as it was
 9955 registered. If, during the call to any such function, a call to the `longjmp()` function is made that
 9956 would terminate the call to the registered function, the behavior is undefined.

9957 If a function registered by a call to `atexit()` fails to return, the remaining registered functions shall
 9958 not be called and the rest of the `exit()` processing shall not be completed. If `exit()` is called more
 9959 than once, the behavior is undefined.

9960 The `exit()` function shall then flush all open streams with unwritten buffered data, close all open
 9961 streams, and remove all files created by `tmpfile()`. Finally, control shall be terminated with the
 9962 consequences described below.

9963 CX The `_Exit()` and `_exit()` functions shall be functionally equivalent.

9964 CX The `_Exit()` and `_exit()` functions shall not call functions registered with `atexit()` nor any
 9965 registered signal handlers. Whether open streams are flushed or closed, or temporary files are
 9966 removed is implementation-defined. Finally, the calling process is terminated with the
 9967 consequences described below.

9968 CX These functions shall terminate the calling process with the following consequences:

9969 **Note:** These consequences are all extensions to the ISO C standard and are not further CX shaded.
 9970 However, XSI extensions are shaded.

9971 XSI • All of the file descriptors, directory streams, conversion descriptors, and message catalog
 9972 descriptors open in the calling process shall be closed.

9973 XSI • If the parent process of the calling process is executing a `wait()` or `waitpid()`, and has neither
 9974 set its `SA_NOCLDWAIT` flag nor set `SIGCHLD` to `SIG_IGN`, it shall be notified of the calling
 9975 process' termination and the low-order eight bits (that is, bits 0377) of *status* are made
 9976 available to it. If the parent is not waiting, the child's status shall be made available to it
 9977 when the parent subsequently executes `wait()` or `waitpid()`.

9978 XSI The semantics of the `waitid()` function shall be equivalent to `wait()`.

9979 XSI • If the parent process of the calling process is not executing a `wait()` or `waitpid()`, and has
 9980 neither set its `SA_NOCLDWAIT` flag nor set `SIGCHLD` to `SIG_IGN`, the calling process shall
 9981 be transformed into a *zombie process*. A *zombie process* is an inactive process and it shall be

9982		deleted at some later time when its parent process executes <code>wait()</code> or <code>waitpid()</code> .
9983	XSI	The semantics of the <code>waitid()</code> function shall be equivalent to <code>wait()</code> .
9984		• Termination of a process does not directly terminate its children. The sending of a SIGHUP signal as described below indirectly terminates children in some circumstances.
9985		
9986		• Either:
9987		If the implementation supports the SIGCHLD signal, a SIGCHLD shall be sent to the parent process.
9988		
9989		Or:
9990	XSI	If the parent process has set its SA_NOCLDWAIT flag, or set SIGCHLD to SIG_IGN, the status shall be discarded, and the lifetime of the calling process shall end immediately. If SA_NOCLDWAIT is set, it is implementation-defined whether a SIGCHLD signal is sent to the parent process.
9991		
9992		
9993		
9994		• The parent process ID of all of the calling process' existing child processes and zombie processes shall be set to the process ID of an implementation-defined system process. That is, these processes shall be inherited by a special system process.
9995		
9996		
9997	XSI	• Each attached shared-memory segment is detached and the value of <code>shm_nattch</code> (see <code>shmget()</code>) in the data structure associated with its shared memory ID shall be decremented by 1.
9998		
9999		
10000	XSI	• For each semaphore for which the calling process has set a <code>semadj</code> value (see <code>semop()</code>), that value shall be added to the <code>semval</code> of the specified semaphore.
10001		
10002		• If the process is a controlling process, the SIGHUP signal shall be sent to each process in the foreground process group of the controlling terminal belonging to the calling process.
10003		
10004		• If the process is a controlling process, the controlling terminal associated with the session shall be disassociated from the session, allowing it to be acquired by a new controlling process.
10005		
10006		
10007		• If the exit of the process causes a process group to become orphaned, and if any member of the newly-orphaned process group is stopped, then a SIGHUP signal followed by a SIGCONT signal shall be sent to each process in the newly-orphaned process group.
10008		
10009		
10010	SEM	• All open named semaphores in the calling process shall be closed as if by appropriate calls to <code>sem_close()</code> .
10011		
10012	ML	• Any memory locks established by the process via calls to <code>mlockall()</code> or <code>mlock()</code> shall be removed. If locked pages in the address space of the calling process are also mapped into the address spaces of other processes and are locked by those processes, the locks established by the other processes shall be unaffected by the call by this process to <code>_Exit()</code> or <code>_exit()</code> .
10013		
10014		
10015		
10016	MF SHM	• Memory mappings that were created in the process shall be unmapped before the process is destroyed.
10017		
10018	TYM	• Any blocks of typed memory that were mapped in the calling process shall be unmapped, as if <code>munmap()</code> was implicitly called to unmap them.
10019		
10020	MSG	• All open message queue descriptors in the calling process shall be closed as if by appropriate calls to <code>mq_close()</code> .
10021		
10022	AIO	• Any outstanding cancelable asynchronous I/O operations may be canceled. Those asynchronous I/O operations that are not canceled shall complete as if the <code>_Exit()</code> or <code>_exit()</code> operation had not yet occurred, but any associated signal notifications shall be suppressed.
10023		
10024		

10025 The `_Exit()` or `_exit()` operation may block awaiting such I/O completion. Whether any I/O
 10026 is canceled, and which I/O may be canceled upon `_Exit()` or `_exit()`, is implementation-
 10027 defined.

10028 • Threads terminated by a call to `_Exit()` or `_exit()` shall not invoke their cancelation cleanup
 10029 handlers or per-thread data destructors.

10030 TRC • If the calling process is a trace controller process, any trace streams that were created by the
 10031 calling process shall be shut down as described by the `posix_trace_shutdown()` function, and
 10032 any process' mapping of trace event names to trace event type identifiers built for these trace
 10033 streams may be deallocated.

10034 RETURN VALUE

10035 These functions do not return.

10036 ERRORS

10037 No errors are defined.

10038 EXAMPLES

10039 None.

10040 APPLICATION USAGE

10041 Normally applications should use `exit()` rather than `_Exit()` or `_exit()`.

10042 RATIONALE

10043 Process Termination

10044 Early proposals drew a distinction between normal and abnormal process termination.
 10045 Abnormal termination was caused only by certain signals and resulted in implementation-
 10046 defined “actions”, as discussed below. Subsequent proposals distinguished three types of
 10047 termination: *normal termination* (as in the current specification), *simple abnormal termination*, and
 10048 *abnormal termination with actions*. Again the distinction between the two types of abnormal
 10049 termination was that they were caused by different signals and that implementation-defined
 10050 actions would result in the latter case. Given that these actions were completely
 10051 implementation-defined, the early proposals were only saying when the actions could occur and
 10052 how their occurrence could be detected, but not what they were. This was of little or no use to
 10053 conforming applications, and thus the distinction is not made in this volume of
 10054 IEEE Std 1003.1-2001.

10055 The implementation-defined actions usually include, in most historical implementations, the
 10056 creation of a file named **core** in the current working directory of the process. This file contains an
 10057 image of the memory of the process, together with descriptive information about the process,
 10058 perhaps sufficient to reconstruct the state of the process at the receipt of the signal.

10059 There is a potential security problem in creating a **core** file if the process was set-user-ID and the
 10060 current user is not the owner of the program, if the process was set-group-ID and none of the
 10061 user's groups match the group of the program, or if the user does not have permission to write in
 10062 the current directory. In this situation, an implementation either should not create a **core** file or
 10063 should make it unreadable by the user.

10064 Despite the silence of this volume of IEEE Std 1003.1-2001 on this feature, applications are
 10065 advised not to create files named **core** because of potential conflicts in many implementations.
 10066 Some implementations use a name other than **core** for the file; for example, by appending the
 10067 process ID to the filename.

Terminating a Process

It is important that the consequences of process termination as described occur regardless of whether the process called `_exit()` (perhaps indirectly through `exit()`) or instead was terminated due to a signal or for some other reason. Note that in the specific case of `exit()` this means that the *status* argument to `exit()` is treated in the same way as the *status* argument to `_exit()`.

A language other than C may have other termination primitives than the C-language `exit()` function, and programs written in such a language should use its native termination primitives, but those should have as part of their function the behavior of `_exit()` as described. Implementations in languages other than C are outside the scope of this version of this volume of IEEE Std 1003.1-2001, however.

As required by the ISO C standard, using **return** from `main()` has the same behavior (other than with respect to language scope issues) as calling `exit()` with the returned value. Reaching the end of the `main()` function has the same behavior as calling `exit(0)`.

A value of zero (or `EXIT_SUCCESS`, which is required to be zero) for the argument *status* conventionally indicates successful termination. This corresponds to the specification for `exit()` in the ISO C standard. The convention is followed by utilities such as *make* and various shells, which interpret a zero status from a child process as success. For this reason, applications should not call `exit(0)` or `_exit(0)` when they terminate unsuccessfully; for example, in signal-catching functions.

Historically, the implementation-defined process that inherits children whose parents have terminated without waiting on them is called *init* and has a process ID of 1.

The sending of a `SIGHUP` to the foreground process group when a controlling process terminates corresponds to somewhat different historical implementations. In System V, the kernel sends a `SIGHUP` on termination of (essentially) a controlling process. In 4.2 BSD, the kernel does not send `SIGHUP` in a case like this, but the termination of a controlling process is usually noticed by a system daemon, which arranges to send a `SIGHUP` to the foreground process group with the `vhangup()` function. However, in 4.2 BSD, due to the behavior of the shells that support job control, the controlling process is usually a shell with no other processes in its process group. Thus, a change to make `_exit()` behave this way in such systems should not cause problems with existing applications.

The termination of a process may cause a process group to become orphaned in either of two ways. The connection of a process group to its parent(s) outside of the group depends on both the parents and their children. Thus, a process group may be orphaned by the termination of the last connecting parent process outside of the group or by the termination of the last direct descendant of the parent process(es). In either case, if the termination of a process causes a process group to become orphaned, processes within the group are disconnected from their job control shell, which no longer has any information on the existence of the process group. Stopped processes within the group would languish forever. In order to avoid this problem, newly orphaned process groups that contain stopped processes are sent a `SIGHUP` signal and a `SIGCONT` signal to indicate that they have been disconnected from their session. The `SIGHUP` signal causes the process group members to terminate unless they are catching or ignoring `SIGHUP`. Under most circumstances, all of the members of the process group are stopped if any of them are stopped.

The action of sending a `SIGHUP` and a `SIGCONT` signal to members of a newly orphaned process group is similar to the action of 4.2 BSD, which sends `SIGHUP` and `SIGCONT` to each stopped child of an exiting process. If such children exit in response to the `SIGHUP`, any additional descendants receive similar treatment at that time. In this volume of IEEE Std 1003.1-2001, the signals are sent to the entire process group at the same time. Also, in

10116 this volume of IEEE Std 1003.1-2001, but not in 4.2 BSD, stopped processes may be orphaned, but
 10117 may be members of a process group that is not orphaned; therefore, the action taken at `_exit()`
 10118 must consider processes other than child processes.

10119 It is possible for a process group to be orphaned by a call to `setpgid()` or `setsid()`, as well as by
 10120 process termination. This volume of IEEE Std 1003.1-2001 does not require sending SIGHUP and
 10121 SIGCONT in those cases, because, unlike process termination, those cases are not caused
 10122 accidentally by applications that are unaware of job control. An implementation can choose to
 10123 send SIGHUP and SIGCONT in those cases as an extension; such an extension must be
 10124 documented as required in `<signal.h>`.

10125 The ISO/IEC 9899:1999 standard adds the `_Exit()` function that results in immediate program
 10126 termination without triggering signals or `atexit()`-registered functions. In IEEE Std 1003.1-2001,
 10127 this is equivalent to the `_exit()` function.

10128 FUTURE DIRECTIONS

10129 None.

10130 SEE ALSO

10131 `atexit()`, `close()`, `fclose()`, `longjmp()`, `posix_trace_shutdown()`, `posix_trace_trid_eventid_open()`,
 10132 `semop()`, `shmget()`, `sigaction()`, `wait()`, `waitid()`, `waitpid()`, the Base Definitions volume of
 10133 IEEE Std 1003.1-2001, `<stdlib.h>`, `<unistd.h>`

10134 CHANGE HISTORY

10135 First released in Issue 1. Derived from Issue 1 of the SVID.

10136 Issue 5

10137 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
 10138 Threads Extension.

10139 Interactions with the SA_NOCLDWAIT flag and SIGCHLD signal are further clarified.

10140 The values of `status` from `exit()` are better described.

10141 Issue 6

10142 Extensions beyond the ISO C standard are marked.

10143 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding semantics for
 10144 typed memory.

10145 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 10146 • The `_Exit()` function is included.
- 10147 • The DESCRIPTION is updated.

10148 The description of tracing semantics is added for alignment with IEEE Std 1003.1q-2000.

10149 References to the `wait3()` function are removed.

10150 **NAME**

10151 exp, expf, expl — exponential function

10152 **SYNOPSIS**

10153 #include <math.h>

10154 double exp(double x);

10155 float expf(float x);

10156 long double expl(long double x);

10157 **DESCRIPTION**

10158 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 10159 conflict between the requirements described here and the ISO C standard is unintentional. This
 10160 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

10161 These functions shall compute the base-*e* exponential of *x*.

10162 An application wishing to check for error situations should set *errno* to zero and call
 10163 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 10164 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 10165 zero, an error has occurred.

10166 **RETURN VALUE**10167 Upon successful completion, these functions shall return the exponential value of *x*.

10168 If the correct value would cause overflow, a range error shall occur and *exp()*, *expf()*, and *expl()*
 10169 shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

10170 If the correct value would cause underflow, and is not representable, a range error may occur,
 10171 **MX** and either 0.0 (if supported), or an implementation-defined value shall be returned.

10172 **MX** If *x* is NaN, a NaN shall be returned.10173 If *x* is ± 0 , 1 shall be returned.10174 If *x* is $-\text{Inf}$, +0 shall be returned.10175 If *x* is $+\text{Inf}$, *x* shall be returned.

10176 If the correct value would cause underflow, and is representable, a range error may occur and
 10177 the correct value shall be returned.

10178 **ERRORS**

10179 These functions shall fail if:

10180 Range Error The result overflows.

10181 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 10182 then *errno* shall be set to [ERANGE]. If the integer expression
 10183 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow
 10184 floating-point exception shall be raised.

10185 These functions may fail if:

10186 Range Error The result underflows.

10187 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 10188 then *errno* shall be set to [ERANGE]. If the integer expression
 10189 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
 10190 floating-point exception shall be raised.

10191 **EXAMPLES**

10192 None.

10193 **APPLICATION USAGE**

10194 Note that for IEEE Std 754-1985 **double**, $709.8 < x$ implies $\exp(x)$ has overflowed. The value
10195 $x < -708.4$ implies $\exp(x)$ has underflowed.

10196 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
10197 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

10198 **RATIONALE**

10199 None.

10200 **FUTURE DIRECTIONS**

10201 None.

10202 **SEE ALSO**

10203 *feclearexcept()*, *fetetestexcept()*, *isnan()*, *log()*, the Base Definitions volume of IEEE Std 1003.1-2001,
10204 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

10205 **CHANGE HISTORY**

10206 First released in Issue 1. Derived from Issue 1 of the SVID.

10207 **Issue 5**

10208 The DESCRIPTION is updated to indicate how an application should check for an error. This
10209 text was previously published in the APPLICATION USAGE section.

10210 **Issue 6**10211 The *expf()* and *expl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

10212 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
10213 revised to align with the ISO/IEC 9899:1999 standard.

10214 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
10215 marked.

10216 **NAME**

10217 exp2, exp2f, exp2l — exponential base 2 functions

10218 **SYNOPSIS**

10219 #include <math.h>

10220 double exp2(double x);

10221 float exp2f(float x);

10222 long double exp2l(long double x);

10223 **DESCRIPTION**

10224 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 10225 conflict between the requirements described here and the ISO C standard is unintentional. This
 10226 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

10227 These functions shall compute the base-2 exponential of *x*.

10228 An application wishing to check for error situations should set *errno* to zero and call
 10229 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 10230 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 10231 zero, an error has occurred.

10232 **RETURN VALUE**10233 Upon successful completion, these functions shall return 2^x .

10234 If the correct value would cause overflow, a range error shall occur and *exp2()*, *exp2f()*, and
 10235 *exp2l()* shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL,
 10236 respectively.

10237 If the correct value would cause underflow, and is not representable, a range error may occur,
 10238 MX and either 0.0 (if supported), or an implementation-defined value shall be returned.

10239 MX If *x* is NaN, a NaN shall be returned.10240 If *x* is ± 0 , 1 shall be returned.10241 If *x* is $-\text{Inf}$, +0 shall be returned.10242 If *x* is $+\text{Inf}$, *x* shall be returned.

10243 If the correct value would cause underflow, and is representable, a range error may occur and
 10244 the correct value shall be returned.

10245 **ERRORS**

10246 These functions shall fail if:

10247 Range Error The result overflows.

10248 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 10249 then *errno* shall be set to [ERANGE]. If the integer expression
 10250 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow
 10251 floating-point exception shall be raised.

10252 These functions may fail if:

10253 Range Error The result underflows.

10254 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 10255 then *errno* shall be set to [ERANGE]. If the integer expression
 10256 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
 10257 floating-point exception shall be raised.

10258 **EXAMPLES**

10259 None.

10260 **APPLICATION USAGE**

10261 For IEEE Std 754-1985 **double**, $1024 \leq x$ implies $\exp2(x)$ has overflowed. The value $x < -1022$
10262 implies $\exp(x)$ has underflowed.

10263 On error, the expressions `(math_errhandling & MATH_ERRNO)` and `(math_errhandling &`
10264 `MATH_ERREXCEPT)` are independent of each other, but at least one of them must be non-zero.

10265 **RATIONALE**

10266 None.

10267 **FUTURE DIRECTIONS**

10268 None.

10269 **SEE ALSO**

10270 *exp()*, *feclearexcept()*, *fetetestexcept()*, *isnan()*, *log()*, the Base Definitions volume of
10271 IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
10272 **<math.h>**

10273 **CHANGE HISTORY**

10274 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

10275 **NAME**

10276 expm1, expm1f, expm1l — compute exponential functions

10277 **SYNOPSIS**

10278 #include <math.h>

10279 double expm1(double x);

10280 float expm1f(float x);

10281 long double expm1l(long double x);

10282 **DESCRIPTION**

10283 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 10284 conflict between the requirements described here and the ISO C standard is unintentional. This
 10285 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

10286 These functions shall compute $e^x - 1.0$.

10287 An application wishing to check for error situations should set *errno* to zero and call
 10288 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 10289 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 10290 zero, an error has occurred.

10291 **RETURN VALUE**10292 Upon successful completion, these functions return $e^x - 1.0$.

10293 If the correct value would cause overflow, a range error shall occur and *expm1()*, *expm1f()*, and
 10294 *expm1l()* shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL,
 10295 respectively.

10296 **MX** If *x* is NaN, a NaN shall be returned.10297 If *x* is ± 0 , ± 0 shall be returned.10298 If *x* is $-\text{Inf}$, -1 shall be returned.10299 If *x* is $+\text{Inf}$, *x* shall be returned.10300 If *x* is subnormal, a range error may occur and *x* should be returned.10301 **ERRORS**

10302 These functions shall fail if:

10303 Range Error The result overflows.

10304 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 10305 then *errno* shall be set to [ERANGE]. If the integer expression
 10306 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow
 10307 floating-point exception shall be raised.

10308 These functions may fail if:

10309 **MX** Range Error The value of *x* is subnormal.

10310 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 10311 then *errno* shall be set to [ERANGE]. If the integer expression
 10312 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
 10313 floating-point exception shall be raised.

10314 **EXAMPLES**

10315 None.

10316 **APPLICATION USAGE**10317 The value of *expm1*(*x*) may be more accurate than *exp*(*x*)−1.0 for small values of *x*.10318 The *expm1*() and *log1p*() functions are useful for financial calculations of $((1+x)^n-1)/x$, namely:10319 `expm1(n * log1p(x))/x`10320 when *x* is very small (for example, when calculating small daily interest rates). These functions
10321 also simplify writing accurate inverse hyperbolic functions.10322 For IEEE Std 754-1985 **double**, $709.8 < x$ implies *expm1*(*x*) has overflowed.10323 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
10324 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.10325 **RATIONALE**

10326 None.

10327 **FUTURE DIRECTIONS**

10328 None.

10329 **SEE ALSO**10330 *exp*(), *feclearexcept*(), *fetestexcept*(), *ilogb*(), *log1p*(), the Base Definitions volume of
10331 IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
10332 <math.h>10333 **CHANGE HISTORY**

10334 First released in Issue 4, Version 2.

10335 **Issue 5**

10336 Moved from X/OPEN UNIX extension to BASE.

10337 **Issue 6**10338 The *expm1f*() and *expm1l*() functions are added for alignment with the ISO/IEC 9899:1999
10339 standard.10340 The *expm1*() function is no longer marked as an extension.10341 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
10342 revised to align with the ISO/IEC 9899:1999 standard.10343 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
10344 marked.

10345 **NAME**

10346 fabs, fabsf, fabsl — absolute value function

10347 **SYNOPSIS**

10348 #include <math.h>

10349 double fabs(double x);

10350 float fabsf(float x);

10351 long double fabsl(long double x);

10352 **DESCRIPTION**

10353 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 10354 conflict between the requirements described here and the ISO C standard is unintentional. This
 10355 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

10356 These functions shall compute the absolute value of their argument x , $|x|$.10357 **RETURN VALUE**10358 Upon successful completion, these functions shall return the absolute value of x .10359 **MX** If x is NaN, a NaN shall be returned.10360 If x is ± 0 , $+0$ shall be returned.10361 If x is $\pm \text{Inf}$, $+\text{Inf}$ shall be returned.10362 **ERRORS**

10363 No errors are defined.

10364 **EXAMPLES**

10365 None.

10366 **APPLICATION USAGE**

10367 None.

10368 **RATIONALE**

10369 None.

10370 **FUTURE DIRECTIONS**

10371 None.

10372 **SEE ALSO**10373 *isnan()*, the Base Definitions volume of IEEE Std 1003.1-2001, <math.h>10374 **CHANGE HISTORY**

10375 First released in Issue 1. Derived from Issue 1 of the SVID.

10376 **Issue 5**

10377 The DESCRIPTION is updated to indicate how an application should check for an error. This
 10378 text was previously published in the APPLICATION USAGE section.

10379 **Issue 6**10380 The *fabsf()* and *fabsl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

10381 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
 10382 revised to align with the ISO/IEC 9899:1999 standard.

10383 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
 10384 marked.

10385 **NAME**

10386 fattach — attach a STREAMS-based file descriptor to a file in the file system name space
 10387 (STREAMS)

10388 **SYNOPSIS**

10389 XSR #include <stropts.h>

10390 int fattach(int *fildes*, const char **path*);

10391

10392 **DESCRIPTION**

10393 The *fattach()* function shall attach a STREAMS-based file descriptor to a file, effectively
 10394 associating a pathname with *fildes*. The application shall ensure that the *fildes* argument is a
 10395 valid open file descriptor associated with a STREAMS file. The *path* argument points to a
 10396 pathname of an existing file. The application shall have the appropriate privileges or be the
 10397 owner of the file named by *path* and have write permission. A successful call to *fattach()* shall
 10398 cause all pathnames that name the file named by *path* to name the STREAMS file associated with
 10399 *fildes*, until the STREAMS file is detached from the file. A STREAMS file can be attached to more
 10400 than one file and can have several pathnames associated with it.

10401 The attributes of the named STREAMS file shall be initialized as follows: the permissions, user
 10402 ID, group ID, and times are set to those of the file named by *path*, the number of links is set to 1,
 10403 and the size and device identifier are set to those of the STREAMS file associated with *fildes*. If
 10404 any attributes of the named STREAMS file are subsequently changed (for example, by *chmod()*),
 10405 neither the attributes of the underlying file nor the attributes of the STREAMS file to which *fildes*
 10406 refers shall be affected.

10407 File descriptors referring to the underlying file, opened prior to an *fattach()* call, shall continue to
 10408 refer to the underlying file.

10409 **RETURN VALUE**

10410 Upon successful completion, *fattach()* shall return 0. Otherwise, *-1* shall be returned and *errno*
 10411 set to indicate the error.

10412 **ERRORS**

10413 The *fattach()* function shall fail if:

- | | | |
|-------|----------------|---|
| 10414 | [EACCES] | Search permission is denied for a component of the path prefix, or the process is the owner of <i>path</i> but does not have write permissions on the file named by <i>path</i> . |
| 10415 | | |
| 10416 | | |
| 10417 | [EBADF] | The <i>fildes</i> argument is not a valid open file descriptor. |
| 10418 | [EBUSY] | The file named by <i>path</i> is currently a mount point or has a STREAMS file attached to it. |
| 10419 | | |
| 10420 | [ELOOP] | A loop exists in symbolic links encountered during resolution of the <i>path</i> argument. |
| 10421 | | |
| 10422 | [ENAMETOOLONG] | |
| 10423 | | The size of <i>path</i> exceeds {PATH_MAX} or a component of <i>path</i> is longer than {NAME_MAX}. |
| 10424 | | |
| 10425 | [ENOENT] | A component of <i>path</i> does not name an existing file or <i>path</i> is an empty string. |
| 10426 | [ENOTDIR] | A component of the path prefix is not a directory. |
| 10427 | [EPERM] | The effective user ID of the process is not the owner of the file named by <i>path</i> and the process does not have appropriate privilege. |
| 10428 | | |

- 10429 The *fattach()* function may fail if:
- 10430 [EINVAL] The *fildev* argument does not refer to a STREAMS file.
 - 10431 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
10432 resolution of the *path* argument.
 - 10433 [ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result
10434 whose length exceeds {PATH_MAX}.
10435
 - 10436 [EXDEV] A link to a file on another file system was attempted.

10437 EXAMPLES

10438 Attaching a File Descriptor to a File

10439 In the following example, *fd* refers to an open STREAMS file. The call to *fattach()* associates this
10440 STREAM with the file **/tmp/named-STREAM**, such that any future calls to open **/tmp/named-**
10441 **STREAM**, prior to breaking the attachment via a call to *fdetach()*, will instead create a new file
10442 handle referring to the STREAMS file associated with *fd*.

```
10443 #include <stropts.h>
10444 ...
10445     int fd;
10446     char *filename = "/tmp/named-STREAM";
10447     int ret;
10448     ret = fattach(fd, filename);
```

10449 APPLICATION USAGE

10450 The *fattach()* function behaves similarly to the traditional *mount()* function in the way a file is
10451 temporarily replaced by the root directory of the mounted file system. In the case of *fattach()*, the
10452 replaced file need not be a directory and the replacing file is a STREAMS file.

10453 RATIONALE

10454 The file attributes of a file which has been the subject of an *fattach()* call are specifically set
10455 because of an artefact of the original implementation. The internal mechanism was the same as
10456 for the *mount()* function. Since *mount()* is typically only applied to directories, the effects when
10457 applied to a regular file are a little surprising, especially as regards the link count which rigidly
10458 remains one, even if there were several links originally and despite the fact that all original links
10459 refer to the STREAM as long as the *fattach()* remains in effect.

10460 FUTURE DIRECTIONS

10461 None.

10462 SEE ALSO

10463 *fdetach()*, *isastream()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**stropts.h**>

10464 CHANGE HISTORY

10465 First released in Issue 4, Version 2.

10466 Issue 5

10467 Moved from X/OPEN UNIX extension to BASE.

10468 The [EXDEV] error is added to the list of optional errors in the ERRORS section.

10469 **Issue 6**

- 10470 This function is marked as part of the XSI STREAMS Option Group.
- 10471 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
- 10472 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
- 10473 [ELOOP] error condition is added.

10474 **NAME**

10475 fchdir — change working directory

10476 **SYNOPSIS**

10477 XSI #include <unistd.h>

10478 int fchdir(int *fil*des);

10479

10480 **DESCRIPTION**

10481 The *fchdir()* function shall be equivalent to *chdir()* except that the directory that is to be the new
 10482 current working directory is specified by the file descriptor *fil*des.

10483 A conforming application can obtain a file descriptor for a file of type directory using *open()*,
 10484 provided that the file status flags and access modes do not contain O_WRONLY or O_RDWR.

10485 **RETURN VALUE**

10486 Upon successful completion, *fchdir()* shall return 0. Otherwise, it shall return -1 and set *errno* to
 10487 indicate the error. On failure the current working directory shall remain unchanged.

10488 **ERRORS**10489 The *fchdir()* function shall fail if:

10490 [EACCES] Search permission is denied for the directory referenced by *fil*des.

10491 [EBADF] The *fil*des argument is not an open file descriptor.

10492 [ENOTDIR] The open file descriptor *fil*des does not refer to a directory.

10493 The *fchdir()* may fail if:

10494 [EINTR] A signal was caught during the execution of *fchdir()*.

10495 [EIO] An I/O error occurred while reading from or writing to the file system.

10496 **EXAMPLES**

10497 None.

10498 **APPLICATION USAGE**

10499 None.

10500 **RATIONALE**

10501 None.

10502 **FUTURE DIRECTIONS**

10503 None.

10504 **SEE ALSO**10505 *chdir()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>10506 **CHANGE HISTORY**

10507 First released in Issue 4, Version 2.

10508 **Issue 5**

10509 Moved from X/OPEN UNIX extension to BASE.

10510 **NAME**

10511 fchmod — change mode of a file

10512 **SYNOPSIS**

10513 #include <sys/stat.h>

10514 int fchmod(int *fildes*, mode_t *mode*);10515 **DESCRIPTION**

10516 The *fchmod()* function shall be equivalent to *chmod()* except that the file whose permissions are
 10517 changed is specified by the file descriptor *fildes*.

10518 SHM If *fildes* references a shared memory object, the *fchmod()* function need only affect the S_IRUSR,
 10519 S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits.

10520 TYM If *fildes* references a typed memory object, the behavior of *fchmod()* is unspecified.

10521 If *fildes* refers to a socket, the behavior of *fchmod()* is unspecified.

10522 XSR If *fildes* refers to a STREAM (which is *fattach()*-ed into the file system name space) the call
 10523 returns successfully, doing nothing.

10524 **RETURN VALUE**

10525 Upon successful completion, *fchmod()* shall return 0. Otherwise, it shall return *-1* and set *errno* to
 10526 indicate the error.

10527 **ERRORS**

10528 The *fchmod()* function shall fail if:

10529 [EBADF] The *fildes* argument is not an open file descriptor.

10530 [EPERM] The effective user ID does not match the owner of the file and the process
 10531 does not have appropriate privilege.

10532 [EROFS] The file referred to by *fildes* resides on a read-only file system.

10533 The *fchmod()* function may fail if:

10534 XSI [EINTR] The *fchmod()* function was interrupted by a signal.

10535 XSI [EINVAL] The value of the *mode* argument is invalid.

10536 [EINVAL] The *fildes* argument refers to a pipe and the implementation disallows
 10537 execution of *fchmod()* on a pipe.

10538 **EXAMPLES**10539 **Changing the Current Permissions for a File**

10540 The following example shows how to change the permissions for a file named */home/cnd/mod1*
 10541 so that the owner and group have read/write/execute permissions, but the world only has
 10542 read/write permissions.

10543 #include <sys/stat.h>

10544 #include <fcntl.h>

10545 mode_t mode;

10546 int fildes;

10547 ...

10548 fildes = open("/home/cnd/mod1", O_RDWR);

10549 fchmod(fildes, S_IRWXU | S_IRWXG | S_IROTH | S_IWOTH);

10550 APPLICATION USAGE

10551 None.

10552 RATIONALE

10553 None.

10554 FUTURE DIRECTIONS

10555 None.

10556 SEE ALSO

10557 *chmod()*, *chown()*, *creat()*, *fcntl()*, *fstatvfs()*, *mknod()*, *open()*, *read()*, *stat()*, *write()*, the Base
10558 Definitions volume of IEEE Std 1003.1-2001, <sys/stat.h>

10559 CHANGE HISTORY

10560 First released in Issue 4, Version 2.

10561 Issue 5

10562 Moved from X/OPEN UNIX extension to BASE and aligned with *fchmod()* in the POSIX
10563 Realtime Extension. Specifically, the second paragraph of the DESCRIPTION is added and a
10564 second instance of [EINVAL] is defined in the list of optional errors.

10565 Issue 6

10566 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by stating that *fchmod()*
10567 behavior is unspecified for typed memory objects.

10568 **NAME**

10569 fchown — change owner and group of a file

10570 **SYNOPSIS**

10571 #include <unistd.h>

10572 int fchown(int *fildev*, uid_t *owner*, gid_t *group*);10573 **DESCRIPTION**

10574 The *fchown()* function shall be equivalent to *chown()* except that the file whose owner and group
 10575 are changed is specified by the file descriptor *fildev*.

10576 **RETURN VALUE**

10577 Upon successful completion, *fchown()* shall return 0. Otherwise, it shall return -1 and set *errno* to
 10578 indicate the error.

10579 **ERRORS**10580 The *fchown()* function shall fail if:10581 [EBADF] The *fildev* argument is not an open file descriptor.

10582 [EPERM] The effective user ID does not match the owner of the file or the process does
 10583 not have appropriate privilege and _POSIX_CHOWN_RESTRICTED indicates
 10584 that such privilege is required.

10585 [EROFS] The file referred to by *fildev* resides on a read-only file system.10586 The *fchown()* function may fail if:

10587 [EINVAL] The owner or group ID is not a value supported by the implementation. The
 10588 *fildev* argument refers to a pipe or socket or an *attach()*-ed STREAM and the
 10589 implementation disallows execution of *fchown()* on a pipe.

10590 [EIO] A physical I/O error has occurred.

10591 [EINTR] The *fchown()* function was interrupted by a signal which was caught.10592 **EXAMPLES**10593 **Changing the Current Owner of a File**

10594 The following example shows how to change the owner of a file named */home/cnd/mod1* to
 10595 “jones” and the group to “cnd”.

10596 The numeric value for the user ID is obtained by extracting the user ID from the user database
 10597 entry associated with “jones”. Similarly, the numeric value for the group ID is obtained by
 10598 extracting the group ID from the group database entry associated with “cnd”. This example
 10599 assumes the calling program has appropriate privileges.

```

10600       #include <sys/types.h>
10601       #include <unistd.h>
10602       #include <fcntl.h>
10603       #include <pwd.h>
10604       #include <grp.h>

10605       struct passwd *pwd;
10606       struct group *grp;
10607       int        fildev;
10608       ...
10609       fildev = open("/home/cnd/mod1", O_RDWR);
10610       pwd = getpwnam("jones");

```



```
10611         grp = getgrnam("cnd");
10612         fchown(fildes, pwd->pw_uid, grp->gr_gid);
```

10613 APPLICATION USAGE

10614 None.

10615 RATIONALE

10616 None.

10617 FUTURE DIRECTIONS

10618 None.

10619 SEE ALSO

10620 *chown()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**unistd.h**>

10621 CHANGE HISTORY

10622 First released in Issue 4, Version 2.

10623 Issue 5

10624 Moved from X/OPEN UNIX extension to BASE.

10625 Issue 6

10626 The following changes were made to align with the IEEE P1003.1a draft standard:

- 10627 • Clarification is added that a call to *fchown()* may not be allowed on a pipe.

10628 The *fchown()* function is now defined as mandatory.

10629 **NAME**10630 `fclose` — close a stream10631 **SYNOPSIS**10632 `#include <stdio.h>`10633 `int fclose(FILE *stream);`10634 **DESCRIPTION**

10635 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 10636 conflict between the requirements described here and the ISO C standard is unintentional. This
 10637 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

10638 The `fclose()` function shall cause the stream pointed to by *stream* to be flushed and the associated
 10639 file to be closed. Any unwritten buffered data for the stream shall be written to the file; any
 10640 unread buffered data shall be discarded. Whether or not the call succeeds, the stream shall be
 10641 disassociated from the file and any buffer set by the `setbuf()` or `setvbuf()` function shall be
 10642 disassociated from the stream. If the associated buffer was automatically allocated, it shall be
 10643 deallocated.

10644 CX The `fclose()` function shall mark for update the `st_ctime` and `st_mtime` fields of the underlying file,
 10645 if the stream was writable, and if buffered data remains that has not yet been written to the file.
 10646 The `fclose()` function shall perform the equivalent of a `close()` on the file descriptor that is
 10647 associated with the stream pointed to by *stream*.

10648 After the call to `fclose()`, any use of *stream* results in undefined behavior.

10649 **RETURN VALUE**

10650 CX Upon successful completion, `fclose()` shall return 0; otherwise, it shall return EOF and set *errno* to
 10651 indicate the error.

10652 **ERRORS**

10653 The `fclose()` function shall fail if:

10654 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the
 10655 process would be delayed in the write operation.

10656 CX [EBADF] The file descriptor underlying stream is not valid.

10657 CX [EFBIG] An attempt was made to write a file that exceeds the maximum file size.

10658 XSI [EFBIG] An attempt was made to write a file that exceeds the process' file size limit.

10659 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
 10660 offset maximum associated with the corresponding stream.

10661 CX [EINTR] The `fclose()` function was interrupted by a signal.

10662 CX [EIO] The process is a member of a background process group attempting to write
 10663 to its controlling terminal, TOSTOP is set, the process is neither ignoring nor
 10664 blocking SIGTTOU, and the process group of the process is orphaned. This
 10665 error may also be returned under implementation-defined conditions.

10666 CX [ENOSPC] There was no free space remaining on the device containing the file.

10667 CX [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by
 10668 any process. A SIGPIPE signal shall also be sent to the thread.

10669 The `fclose()` function may fail if:

10670 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
 10671 capabilities of the device.

10672 **EXAMPLES**

10673 None.

10674 **APPLICATION USAGE**

10675 None.

10676 **RATIONALE**

10677 None.

10678 **FUTURE DIRECTIONS**

10679 None.

10680 **SEE ALSO**

10681 *close()*, *fopen()*, *getrlimit()*, *ulimit()*, the Base Definitions volume of IEEE Std 1003.1-2001,
10682 <stdio.h>

10683 **CHANGE HISTORY**

10684 First released in Issue 1. Derived from Issue 1 of the SVID.

10685 **Issue 5**

10686 Large File Summit extensions are added.

10687 **Issue 6**

10688 Extensions beyond the ISO C standard are marked.

10689 The following new requirements on POSIX implementations derive from alignment with the
10690 Single UNIX Specification:

- 10691 • The [EFBIG] error is added as part of the large file support extensions.
- 10692 • The [ENXIO] optional error condition is added.

10693 The DESCRIPTION is updated to note that the stream and any buffer are disassociated whether
10694 or not the call succeeds. This is for alignment with the ISO/IEC 9899:1999 standard.

10695 **NAME**

10696 fcntl — file control

10697 **SYNOPSIS**

10698 OH #include <unistd.h>

10699 #include <fcntl.h>

10700 int fcntl(int *fildes*, int *cmd*, ...);10701 **DESCRIPTION**

10702 The *fcntl()* function shall perform the operations described below on open files. The *fildes*
 10703 argument is a file descriptor.

10704 The available values for *cmd* are defined in <fcntl.h> and are as follows:

10705 **F_DUPFD** Return a new file descriptor which shall be the lowest numbered available
 10706 (that is, not already open) file descriptor greater than or equal to the third
 10707 argument, *arg*, taken as an integer of type **int**. The new file descriptor shall
 10708 refer to the same open file description as the original file descriptor, and shall
 10709 share any locks. The FD_CLOEXEC flag associated with the new file
 10710 descriptor shall be cleared to keep the file open across calls to one of the *exec*
 10711 functions.

10712 **F_GETFD** Get the file descriptor flags defined in <fcntl.h> that are associated with the
 10713 file descriptor *fildes*. File descriptor flags are associated with a single file
 10714 descriptor and do not affect other file descriptors that refer to the same file.

10715 **F_SETFD** Set the file descriptor flags defined in <fcntl.h>, that are associated with *fildes*,
 10716 to the third argument, *arg*, taken as type **int**. If the FD_CLOEXEC flag in the
 10717 third argument is 0, the file shall remain open across the *exec* functions;
 10718 otherwise, the file shall be closed upon successful execution of one of the *exec*
 10719 functions.

10720 **F_GETFL** Get the file status flags and file access modes, defined in <fcntl.h>, for the file
 10721 description associated with *fildes*. The file access modes can be extracted from
 10722 the return value using the mask O_ACCMODE, which is defined in <fcntl.h>.
 10723 File status flags and file access modes are associated with the file description
 10724 and do not affect other file descriptors that refer to the same file with different
 10725 open file descriptions.

10726 **F_SETFL** Set the file status flags, defined in <fcntl.h>, for the file description associated
 10727 with *fildes* from the corresponding bits in the third argument, *arg*, taken as
 10728 type **int**. Bits corresponding to the file access mode and the file creation flags,
 10729 as defined in <fcntl.h>, that are set in *arg* shall be ignored. If any bits in *arg*
 10730 other than those mentioned here are changed by the application, the result is
 10731 unspecified.

10732 **F_GETOWN** If *fildes* refers to a socket, get the process or process group ID specified to
 10733 receive SIGURG signals when out-of-band data is available. Positive values
 10734 indicate a process ID; negative values, other than -1, indicate a process group
 10735 ID. If *fildes* does not refer to a socket, the results are unspecified.

10736 **F_SETOWN** If *fildes* refers to a socket, set the process or process group ID specified to
 10737 receive SIGURG signals when out-of-band data is available, using the value of
 10738 the third argument, *arg*, taken as type **int**. Positive values indicate a process
 10739 ID; negative values, other than -1, indicate a process group ID. If *fildes* does
 10740 not refer to a socket, the results are unspecified.

10741	The following values for <i>cmd</i> are available for advisory record locking. Record locking shall be	
10742	supported for regular files, and may be supported for other files.	
10743	F_GETLK	Get the first lock which blocks the lock description pointed to by the third
10744		argument, <i>arg</i> , taken as a pointer to type struct flock , defined in <code><fcntl.h></code> .
10745		The information retrieved shall overwrite the information passed to <i>fcntl()</i> in
10746		the structure flock . If no lock is found that would prevent this lock from
10747		being created, then the structure shall be left unchanged except for the lock
10748		type which shall be set to F_UNLCK.
10749	F_SETLK	Set or clear a file segment lock according to the lock description pointed to by
10750		the third argument, <i>arg</i> , taken as a pointer to type struct flock , defined in
10751		<code><fcntl.h></code> . F_SETLK can establish shared (or read) locks (F_RDLCK) or
10752		exclusive (or write) locks (F_WRLCK), as well as to remove either type of lock
10753		(F_UNLCK). F_RDLCK, F_WRLCK, and F_UNLCK are defined in <code><fcntl.h></code> .
10754		If a shared or exclusive lock cannot be set, <i>fcntl()</i> shall return immediately
10755		with a return value of <code>-1</code> .
10756	F_SETLKW	This command shall be equivalent to F_SETLK except that if a shared or
10757		exclusive lock is blocked by other locks, the thread shall wait until the request
10758		can be satisfied. If a signal that is to be caught is received while <i>fcntl()</i> is
10759		waiting for a region, <i>fcntl()</i> shall be interrupted. Upon return from the signal
10760		handler, <i>fcntl()</i> shall return <code>-1</code> with <i>errno</i> set to [EINTR], and the lock
10761		operation shall not be done.
10762	Additional implementation-defined values for <i>cmd</i> may be defined in <code><fcntl.h></code> . Their names	
10763	shall start with F_.	
10764	When a shared lock is set on a segment of a file, other processes shall be able to set shared locks	
10765	on that segment or a portion of it. A shared lock prevents any other process from setting an	
10766	exclusive lock on any portion of the protected area. A request for a shared lock shall fail if the	
10767	file descriptor was not opened with read access.	
10768	An exclusive lock shall prevent any other process from setting a shared lock or an exclusive lock	
10769	on any portion of the protected area. A request for an exclusive lock shall fail if the file	
10770	descriptor was not opened with write access.	
10771	The structure flock describes the type (<i>l_type</i>), starting offset (<i>l_whence</i>), relative offset (<i>l_start</i>),	
10772	size (<i>l_len</i>), and process ID (<i>l_pid</i>) of the segment of the file to be affected.	
10773	The value of <i>l_whence</i> is SEEK_SET, SEEK_CUR, or SEEK_END, to indicate that the relative	
10774	offset <i>l_start</i> bytes shall be measured from the start of the file, current position, or end of the file,	
10775	respectively. The value of <i>l_len</i> is the number of consecutive bytes to be locked. The value of	
10776	<i>l_len</i> may be negative (where the definition of off_t permits negative values of <i>l_len</i>). The <i>l_pid</i>	
10777	field is only used with F_GETLK to return the process ID of the process holding a blocking lock.	
10778	After a successful F_GETLK request, when a blocking lock is found, the values returned in the	
10779	flock structure shall be as follows:	
10780	<i>l_type</i>	Type of blocking lock found.
10781	<i>l_whence</i>	SEEK_SET.
10782	<i>l_start</i>	Start of the blocking lock.
10783	<i>l_len</i>	Length of the blocking lock.
10784	<i>l_pid</i>	Process ID of the process that holds the blocking lock.

10785		If the command is F_SETLKW and the process must wait for another process to release a lock,
10786		then the range of bytes to be locked shall be determined before the <i>fcntl()</i> function blocks. If the
10787		file size or file descriptor seek offset change while <i>fcntl()</i> is blocked, this shall not affect the
10788		range of bytes locked.
10789		If <i>l_len</i> is positive, the area affected shall start at <i>l_start</i> and end at <i>l_start+l_len-1</i> . If <i>l_len</i> is
10790		negative, the area affected shall start at <i>l_start+l_len</i> and end at <i>l_start-1</i> . Locks may start and
10791		extend beyond the current end of a file, but shall not extend before the beginning of the file. A
10792		lock shall be set to extend to the largest possible value of the file offset for that file by setting
10793		<i>l_len</i> to 0. If such a lock also has <i>l_start</i> set to 0 and <i>l_whence</i> is set to SEEK_SET, the whole file
10794		shall be locked.
10795		There shall be at most one type of lock set for each byte in the file. Before a successful return
10796		from an F_SETLK or an F_SETLKW request when the calling process has previously existing
10797		locks on bytes in the region specified by the request, the previous lock type for each byte in the
10798		specified region shall be replaced by the new lock type. As specified above under the
10799		descriptions of shared locks and exclusive locks, an F_SETLK or an F_SETLKW request
10800		(respectively) shall fail or block when another process has existing locks on bytes in the specified
10801		region and the type of any of those locks conflicts with the type specified in the request.
10802		All locks associated with a file for a given process shall be removed when a file descriptor for
10803		that file is closed by that process or the process holding that file descriptor terminates. Locks are
10804		not inherited by a child process.
10805		A potential for deadlock occurs if a process controlling a locked region is put to sleep by
10806		attempting to lock another process' locked region. If the system detects that sleeping until a
10807		locked region is unlocked would cause a deadlock, <i>fcntl()</i> shall fail with an [EDEADLK] error.
10808		An unlock (F_UNLCK) request in which <i>l_len</i> is non-zero and the offset of the last byte of the
10809		requested segment is the maximum value for an object of type <i>off_t</i> , when the process has an
10810		existing lock in which <i>l_len</i> is 0 and which includes the last byte of the requested segment, shall
10811		be treated as a request to unlock from the start of the requested segment with an <i>l_len</i> equal to 0.
10812		Otherwise, an unlock (F_UNLCK) request shall attempt to unlock only the requested segment.
10813	SHM	When the file descriptor <i>fdes</i> refers to a shared memory object, the behavior of <i>fcntl()</i> shall be
10814		the same as for a regular file except the effect of the following values for the argument <i>cmd</i> shall
10815		be unspecified: F_SETFL, F_GETLK, F_SETLK, and F_SETLKW.
10816	TYM	If <i>fdes</i> refers to a typed memory object, the result of the <i>fcntl()</i> function is unspecified.
10817	RETURN VALUE	
10818		Upon successful completion, the value returned shall depend on <i>cmd</i> as follows:
10819	F_DUPFD	A new file descriptor.
10820	F_GETFD	Value of flags defined in <fcntl.h>. The return value shall not be negative.
10821	F_SETFD	Value other than -1.
10822	F_GETFL	Value of file status flags and access modes. The return value is not negative.
10823	F_SETFL	Value other than -1.
10824	F_GETLK	Value other than -1.
10825	F_SETLK	Value other than -1.
10826	F_SETLKW	Value other than -1.
10827	F_GETOWN	Value of the socket owner process or process group; this will not be -1.

10828	F_SETOWN	Value other than -1.
10829		Otherwise, -1 shall be returned and <i>errno</i> set to indicate the error.
10830	ERRORS	
10831		The <i>fcntl()</i> function shall fail if:
10832	[EACCES] or [EAGAIN]	
10833		The <i>cmd</i> argument is F_SETLK; the type of lock (<i>l_type</i>) is a shared (F_RDLCK) or exclusive (F_WRLCK) lock and the segment of a file to be locked is already exclusive-locked by another process, or the type is an exclusive lock and some portion of the segment of a file to be locked is already shared-locked or exclusive-locked by another process.
10834		
10835		
10836		
10837		
10838	[EBADF]	The <i>fildev</i> argument is not a valid open file descriptor, or the argument <i>cmd</i> is F_SETLK or F_SETLKW, the type of lock, <i>l_type</i> , is a shared lock (F_RDLCK), and <i>fildev</i> is not a valid file descriptor open for reading, or the type of lock, <i>l_type</i> , is an exclusive lock (F_WRLCK), and <i>fildev</i> is not a valid file descriptor open for writing.
10839		
10840		
10841		
10842		
10843	[EINTR]	The <i>cmd</i> argument is F_SETLKW and the function was interrupted by a signal.
10844	[EINVAL]	The <i>cmd</i> argument is invalid, or the <i>cmd</i> argument is F_DUPFD and <i>arg</i> is negative or greater than or equal to {OPEN_MAX}, or the <i>cmd</i> argument is F_GETLK, F_SETLK, or F_SETLKW and the data pointed to by <i>arg</i> is not valid, or <i>fildev</i> refers to a file that does not support locking.
10845		
10846		
10847		
10848	[EMFILE]	The argument <i>cmd</i> is F_DUPFD and {OPEN_MAX} file descriptors are currently open in the calling process, or no file descriptors greater than or equal to <i>arg</i> are available.
10849		
10850		
10851	[ENOLCK]	The argument <i>cmd</i> is F_SETLK or F_SETLKW and satisfying the lock or unlock request would result in the number of locked regions in the system exceeding a system-imposed limit.
10852		
10853		
10854	[EOVERFLOW]	One of the values to be returned cannot be represented correctly.
10855	[EOVERFLOW]	The <i>cmd</i> argument is F_GETLK, F_SETLK, or F_SETLKW and the smallest or, if <i>l_len</i> is non-zero, the largest offset of any byte in the requested segment cannot be represented correctly in an object of type off_t .
10856		
10857		
10858		The <i>fcntl()</i> function may fail if:
10859	[EDEADLK]	The <i>cmd</i> argument is F_SETLKW, the lock is blocked by a lock from another process, and putting the calling process to sleep to wait for that lock to become free would cause a deadlock.
10860		
10861		

10862 **EXAMPLES**

10863 None.

10864 **APPLICATION USAGE**

10865 None.

10866 **RATIONALE**

10867 The ellipsis in the SYNOPSIS is the syntax specified by the ISO C standard for a variable number
 10868 of arguments. It is used because System V uses pointers for the implementation of file locking
 10869 functions.

10870 The *arg* values to F_GETFD, F_SETFD, F_GETFL, and F_SETFL all represent flag values to allow
 10871 for future growth. Applications using these functions should do a read-modify-write operation

10872 on them, rather than assuming that only the values defined by this volume of
10873 IEEE Std 1003.1-2001 are valid. It is a common error to forget this, particularly in the case of
10874 F_SETFD.

10875 This volume of IEEE Std 1003.1-2001 permits concurrent read and write access to file data using
10876 the *fcntl()* function; this is a change from the 1984 /usr/group standard and early proposals.
10877 Without concurrency controls, this feature may not be fully utilized without occasional loss of
10878 data.

10879 Data losses occur in several ways. One case occurs when several processes try to update the
10880 same record, without sequencing controls; several updates may occur in parallel and the last
10881 writer “wins”. Another case is a bit-tree or other internal list-based database that is undergoing
10882 reorganization. Without exclusive use to the tree segment by the updating process, other reading
10883 processes chance getting lost in the database when the index blocks are split, condensed,
10884 inserted, or deleted. While *fcntl()* is useful for many applications, it is not intended to be overly
10885 general and does not handle the bit-tree example well.

10886 This facility is only required for regular files because it is not appropriate for many devices such
10887 as terminals and network connections.

10888 Since *fcntl()* works with “any file descriptor associated with that file, however it is obtained”,
10889 the file descriptor may have been inherited through a *fork()* or *exec* operation and thus may
10890 affect a file that another process also has open.

10891 The use of the open file description to identify what to lock requires extra calls and presents
10892 problems if several processes are sharing an open file description, but there are too many
10893 implementations of the existing mechanism for this volume of IEEE Std 1003.1-2001 to use
10894 different specifications.

10895 Another consequence of this model is that closing any file descriptor for a given file (whether or
10896 not it is the same open file description that created the lock) causes the locks on that file to be
10897 relinquished for that process. Equivalently, any close for any file/process pair relinquishes the
10898 locks owned on that file for that process. But note that while an open file description may be
10899 shared through *fork()*, locks are not inherited through *fork()*. Yet locks may be inherited through
10900 one of the *exec* functions.

10901 The identification of a machine in a network environment is outside the scope of this volume of
10902 IEEE Std 1003.1-2001. Thus, an *L_sysid* member, such as found in System V, is not included in the
10903 locking structure.

10904 Changing of lock types can result in a previously locked region being split into smaller regions.

10905 Mandatory locking was a major feature of the 1984 /usr/group standard.

10906 For advisory file record locking to be effective, all processes that have access to a file must
10907 cooperate and use the advisory mechanism before doing I/O on the file. Enforcement-mode
10908 record locking is important when it cannot be assumed that all processes are cooperating. For
10909 example, if one user uses an editor to update a file at the same time that a second user executes
10910 another process that updates the same file and if only one of the two processes is using advisory
10911 locking, the processes are not cooperating. Enforcement-mode record locking would protect
10912 against accidental collisions.

10913 Secondly, advisory record locking requires a process using locking to bracket each I/O operation
10914 with lock (or test) and unlock operations. With enforcement-mode file and record locking, a
10915 process can lock the file once and unlock when all I/O operations have been completed.
10916 Enforcement-mode record locking provides a base that can be enhanced; for example, with
10917 sharable locks. That is, the mechanism could be enhanced to allow a process to lock a file so
10918 other processes could read it, but none of them could write it.

10919 Mandatory locks were omitted for several reasons:

- 10920 1. Mandatory lock setting was done by multiplexing the set-group-ID bit in most
10921 implementations; this was confusing, at best.
- 10922 2. The relationship to file truncation as supported in 4.2 BSD was not well specified.
- 10923 3. Any publicly readable file could be locked by anyone. Many historical implementations
10924 keep the password database in a publicly readable file. A malicious user could thus
10925 prohibit logins. Another possibility would be to hold open a long-distance telephone line.
- 10926 4. Some demand-paged historical implementations offer memory mapped files, and
10927 enforcement cannot be done on that type of file.

10928 Since sleeping on a region is interrupted with any signal, *alarm()* may be used to provide a
10929 timeout facility in applications requiring it. This is useful in deadlock detection. Since
10930 implementation of full deadlock detection is not always feasible, the [EDEADLK] error was
10931 made optional.

10932 FUTURE DIRECTIONS

10933 None.

10934 SEE ALSO

10935 *alarm()*, *close()*, *exec*, *open()*, *sigaction()*, the Base Definitions volume of IEEE Std 1003.1-2001,
10936 *<fcntl.h>*, *<signal.h>*, *<unistd.h>*

10937 CHANGE HISTORY

10938 First released in Issue 1. Derived from Issue 1 of the SVID.

10939 Issue 5

10940 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
10941 Threads Extension.

10942 Large File Summit extensions are added.

10943 Issue 6

10944 In the SYNOPSIS, the optional include of the *<sys/types.h>* header is removed.

10945 The following new requirements on POSIX implementations derive from alignment with the
10946 Single UNIX Specification:

- 10947 • The requirement to include *<sys/types.h>* has been removed. Although *<sys/types.h>* was
10948 required for conforming implementations of previous POSIX specifications, it was not
10949 required for UNIX applications.
- 10950 • In the DESCRIPTION, sentences describing behavior when *l_len* is negative are now
10951 mandated, and the description of unlock (F_UNLOCK) when *l_len* is non-negative is
10952 mandated.
- 10953 • In the ERRORS section, the [EINVAL] error condition has the case mandated when the *cmd* is
10954 invalid, and two [EOVERFLOW] error conditions are added.

10955 The F_GETOWN and F_SETOWN values are added for sockets.

10956 The following changes were made to align with the IEEE P1003.1a draft standard:

- 10957 • Clarification is added that the extent of the bytes locked is determined prior to the blocking
10958 action.

10959 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that
10960 *fcntl()* results are unspecified for typed memory objects.

10961

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

10962 **NAME**10963 fcvt — convert a floating-point number to a string (**LEGACY**)10964 **SYNOPSIS**

10965 XSI #include <stdlib.h>

10966 char *fcvt(double *value*, int *ndigit*, int *restrict *decpt*,
10967 int *restrict *sign*);

10968

10969 **DESCRIPTION**10970 Refer to *ecvt()*.

10971 **NAME**10972 *fdatasync* — synchronize the data of a file (**REALTIME**)10973 **SYNOPSIS**10974 SIO `#include <unistd.h>`10975 `int fdatasync(int fildes);`

10976

10977 **DESCRIPTION**10978 The *fdatasync()* function shall force all currently queued I/O operations associated with the file
10979 indicated by file descriptor *fil*des to the synchronized I/O completion state.10980 The functionality shall be equivalent to *fsync()* with the symbol `_POSIX_SYNCHRONIZED_IO`
10981 defined, with the exception that all I/O operations shall be completed as defined for
10982 synchronized I/O data integrity completion.10983 **RETURN VALUE**10984 If successful, the *fdatasync()* function shall return the value 0; otherwise, the function shall return
10985 the value `-1` and set *errno* to indicate the error. If the *fdatasync()* function fails, outstanding I/O
10986 operations are not guaranteed to have been completed.10987 **ERRORS**10988 The *fdatasync()* function shall fail if:10989 [EBADF] The *fil*des argument is not a valid file descriptor open for writing.

10990 [EINVAL] This implementation does not support synchronized I/O for this file.

10991 In the event that any of the queued I/O operations fail, *fdatasync()* shall return the error
10992 conditions defined for *read()* and *write()*.10993 **EXAMPLES**

10994 None.

10995 **APPLICATION USAGE**

10996 None.

10997 **RATIONALE**

10998 None.

10999 **FUTURE DIRECTIONS**

11000 None.

11001 **SEE ALSO**11002 *aio_fsync()*, *fcntl()*, *fsync()*, *open()*, *read()*, *write()*, the Base Definitions volume of
11003 IEEE Std 1003.1-2001, `<unistd.h>`11004 **CHANGE HISTORY**

11005 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

11006 **Issue 6**11007 The [ENOSYS] error condition has been removed as stubs need not be provided if an
11008 implementation does not support the Synchronized Input and Output option.11009 The *fdatasync()* function is marked as part of the Synchronized Input and Output option.

11010 **NAME**11011 fdetach — detach a name from a STREAMS-based file descriptor (**STREAMS**)11012 **SYNOPSIS**

11013 XSR #include <stropts.h>

11014 int fdetach(const char *path);

11015

11016 **DESCRIPTION**

11017 The *fdetach()* function shall detach a STREAMS-based file from the file to which it was attached
 11018 by a previous call to *fattach()*. The *path* argument points to the pathname of the attached
 11019 STREAMS file. The process shall have appropriate privileges or be the owner of the file. A
 11020 successful call to *fdetach()* shall cause all pathnames that named the attached STREAMS file to
 11021 again name the file to which the STREAMS file was attached. All subsequent operations on *path*
 11022 shall operate on the underlying file and not on the STREAMS file.

11023 All open file descriptions established while the STREAMS file was attached to the file referenced
 11024 by *path* shall still refer to the STREAMS file after the *fdetach()* has taken effect.

11025 If there are no open file descriptors or other references to the STREAMS file, then a successful
 11026 call to *fdetach()* shall be equivalent to performing the last *close()* on the attached file.

11027 **RETURN VALUE**

11028 Upon successful completion, *fdetach()* shall return 0; otherwise, it shall return -1 and set *errno* to
 11029 indicate the error.

11030 **ERRORS**11031 The *fdetach()* function shall fail if:

11032 [EACCES] Search permission is denied on a component of the path prefix.

11033 [EINVAL] The *path* argument names a file that is not currently attached.

11034 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 11035 argument.

11036 [ENAMETOOLONG]

11037 The size of a pathname exceeds {PATH_MAX} or a pathname component is
 11038 longer than {NAME_MAX}.

11039 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

11040 [ENOTDIR] A component of the path prefix is not a directory.

11041 [EPERM] The effective user ID is not the owner of *path* and the process does not have
 11042 appropriate privileges.

11043 The *fdetach()* function may fail if:

11044 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 11045 resolution of the *path* argument.

11046 [ENAMETOOLONG]

11047 Pathname resolution of a symbolic link produced an intermediate result
 11048 whose length exceeds {PATH_MAX}.

11049 **EXAMPLES**11050 **Detaching a File**

11051 The following example detaches the STREAMS-based file **/tmp/named-STREAM** from the file to
11052 which it was attached by a previous, successful call to *fattach()*. Subsequent calls to open this
11053 file refer to the underlying file, not to the STREAMS file.

```
11054 #include <stropts.h>
11055 ...
11056     char *filename = "/tmp/named-STREAM";
11057     int ret;
11058     ret = fdetach(filename);
```

11059 **APPLICATION USAGE**

11060 None.

11061 **RATIONALE**

11062 None.

11063 **FUTURE DIRECTIONS**

11064 None.

11065 **SEE ALSO**

11066 *fattach()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stropts.h>

11067 **CHANGE HISTORY**

11068 First released in Issue 4, Version 2.

11069 **Issue 5**

11070 Moved from X/OPEN UNIX extension to BASE.

11071 **Issue 6**

11072 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

11073 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
11074 [ELOOP] error condition is added.

11075 **NAME**

11076 `fdim`, `fdimf`, `fdiml` — compute positive difference between two floating-point numbers

11077 **SYNOPSIS**

11078 `#include <math.h>`

11079 `double fdim(double x, double y);`

11080 `float fdimf(float x, float y);`

11081 `long double fdiml(long double x, long double y);`

11082 **DESCRIPTION**

11083 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 11084 conflict between the requirements described here and the ISO C standard is unintentional. This
 11085 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11086 These functions shall determine the positive difference between their arguments. If *x* is greater
 11087 than *y*, *x*−*y* is returned. If *x* is less than or equal to *y*, +0 is returned.

11088 An application wishing to check for error situations should set *errno* to zero and call
 11089 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 11090 *fetetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 11091 zero, an error has occurred.

11092 **RETURN VALUE**

11093 Upon successful completion, these functions shall return the positive difference value.

11094 If *x*−*y* is positive and overflows, a range error shall occur and *fdim*(), *fdimf*(), and *fdiml*() shall
 11095 return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

11096 **XSI** If *x*−*y* is positive and underflows, a range error may occur, and either (*x*−*y*) (if representable), or
 11097 0.0 (if supported), or an implementation-defined value shall be returned.

11098 **MX** If *x* or *y* is NaN, a NaN shall be returned.

11099 **ERRORS**

11100 The *fdim*() function shall fail if:

11101 **Range Error** The result overflows.

11102 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 11103 then *errno* shall be set to [ERANGE]. If the integer expression
 11104 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow
 11105 floating-point exception shall be raised.

11106 The *fdim*() function may fail if:

11107 **Range Error** The result underflows.

11108 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 11109 then *errno* shall be set to [ERANGE]. If the integer expression
 11110 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
 11111 floating-point exception shall be raised.

11112 EXAMPLES

11113 None.

11114 APPLICATION USAGE

11115 On implementations supporting IEEE Std 754-1985, $x-y$ cannot underflow, and hence the 0.0
11116 return value is shaded as an extension for implementations supporting the XSI extension rather
11117 than an MX extension.

11118 On error, the expressions `(math_errhandling & MATH_ERRNO)` and `(math_errhandling &`
11119 `MATH_ERREXCEPT)` are independent of each other, but at least one of them must be non-zero.

11120 RATIONALE

11121 None.

11122 FUTURE DIRECTIONS

11123 None.

11124 SEE ALSO

11125 *feclearexcept()*, *fetestexcept()*, *fmax()*, *fmin()*, the Base Definitions volume of IEEE Std 1003.1-2001,
11126 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

11127 CHANGE HISTORY

11128 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

11129 **NAME**

11130 fdopen — associate a stream with a file descriptor

11131 **SYNOPSIS**11132 CX `#include <stdio.h>`11133 `FILE *fdopen(int fildes, const char *mode);`

11134

11135 **DESCRIPTION**11136 The *fdopen()* function shall associate a stream with a file descriptor.11137 The *mode* argument is a character string having one of the following values:11138 *r* or *rb* Open a file for reading.11139 *w* or *wb* Open a file for writing.11140 *a* or *ab* Open a file for writing at end-of-file.11141 *r+* or *rb+* or *r+b* Open a file for update (reading and writing).11142 *w+* or *wb+* or *w+b* Open a file for update (reading and writing).11143 *a+* or *ab+* or *a+b* Open a file for update (reading and writing) at end-of-file.11144 The meaning of these flags is exactly as specified in *fopen()*, except that modes beginning with *w*
11145 shall not cause truncation of the file.11146 Additional values for the *mode* argument may be supported by an implementation.11147 The application shall ensure that the mode of the stream as expressed by the *mode* argument is
11148 allowed by the file access mode of the open file description to which *fildes* refers. The file
11149 position indicator associated with the new stream is set to the position indicated by the file
11150 offset associated with the file descriptor.11151 The error and end-of-file indicators for the stream shall be cleared. The *fdopen()* function may
11152 cause the *st_atime* field of the underlying file to be marked for update.11153 SHM If *fildes* refers to a shared memory object, the result of the *fdopen()* function is unspecified.11154 TYM If *fildes* refers to a typed memory object, the result of the *fdopen()* function is unspecified.11155 The *fdopen()* function shall preserve the offset maximum previously set for the open file
11156 description corresponding to *fildes*.11157 **RETURN VALUE**11158 Upon successful completion, *fdopen()* shall return a pointer to a stream; otherwise, a null pointer
11159 shall be returned and *errno* set to indicate the error.11160 **ERRORS**11161 The *fdopen()* function may fail if:11162 [EBADF] The *fildes* argument is not a valid file descriptor.11163 [EINVAL] The *mode* argument is not a valid mode.

11164 [EMFILE] {FOPEN_MAX} streams are currently open in the calling process.

11165 [EMFILE] {STREAM_MAX} streams are currently open in the calling process.

11166 [ENOMEM] Insufficient space to allocate a buffer.

11167 **EXAMPLES**

11168 None.

11169 **APPLICATION USAGE**

11170 File descriptors are obtained from calls like *open()*, *dup()*, *creat()*, or *pipe()*, which open files but
 11171 do not return streams.

11172 **RATIONALE**

11173 The file descriptor may have been obtained from *open()*, *creat()*, *pipe()*, *dup()*, or *fcntl()*;
 11174 inherited through *fork()* or *exec*; or perhaps obtained by implementation-defined means, such as
 11175 the 4.3 BSD *socket()* call.

11176 The meanings of the *mode* arguments of *fdopen()* and *fopen()* differ. With *fdopen()*, open for write
 11177 (*w* or *w+*) does not truncate, and append (*a* or *a+*) cannot create for writing. The *mode* argument
 11178 formats that include *a b* are allowed for consistency with the ISO C standard function *fopen()*.
 11179 The *b* has no effect on the resulting stream. Although not explicitly required by this volume of
 11180 IEEE Std 1003.1-2001, a good implementation of append (*a*) mode would cause the *O_APPEND*
 11181 flag to be set.

11182 **FUTURE DIRECTIONS**

11183 None.

11184 **SEE ALSO**

11185 Section 2.5.1 (on page 35), *fclose()*, *fopen()*, *open()*, the Base Definitions volume of
 11186 IEEE Std 1003.1-2001, <stdio.h>

11187 **CHANGE HISTORY**

11188 First released in Issue 1. Derived from Issue 1 of the SVID.

11189 **Issue 5**

11190 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

11191 Large File Summit extensions are added.

11192 **Issue 6**

11193 The following new requirements on POSIX implementations derive from alignment with the
 11194 Single UNIX Specification:

- 11195 • In the DESCRIPTION, the use and setting of the *mode* argument are changed to include
 11196 binary streams.
- 11197 • In the DESCRIPTION, text is added for large file support to indicate setting of the offset
 11198 maximum in the open file description.
- 11199 • All errors identified in the ERRORS section are added.
- 11200 • In the DESCRIPTION, text is added that the *fdopen()* function may cause *st_atime* to be
 11201 updated.

11202 The following changes were made to align with the IEEE P1003.1a draft standard:

- 11203 • Clarification is added that it is the responsibility of the application to ensure that the mode is
 11204 compatible with the open file descriptor.

11205 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that
 11206 *fdopen()* results are unspecified for typed memory objects.

11207 **NAME**

11208 feclearexcept — clear floating-point exception

11209 **SYNOPSIS**

11210 #include <fenv.h>

11211 int feclearexcept(int *excepts*);11212 **DESCRIPTION**

11213 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
11214 conflict between the requirements described here and the ISO C standard is unintentional. This
11215 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11216 The *feclearexcept()* function shall attempt to clear the supported floating-point exceptions
11217 represented by *excepts*.

11218 **RETURN VALUE**

11219 If the argument is zero or if all the specified exceptions were successfully cleared, *feclearexcept()*
11220 shall return zero. Otherwise, it shall return a non-zero value.

11221 **ERRORS**

11222 No errors are defined.

11223 **EXAMPLES**

11224 None.

11225 **APPLICATION USAGE**

11226 None.

11227 **RATIONALE**

11228 None.

11229 **FUTURE DIRECTIONS**

11230 None.

11231 **SEE ALSO**

11232 *fegetexceptflag()*, *feraiseexcept()*, *fesetexceptflag()*, *fetestexcept()*, the Base Definitions volume of
11233 IEEE Std 1003.1-2001, <fenv.h>

11234 **CHANGE HISTORY**

11235 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

11236 ISO/IEC 9899:1999 standard, Technical Corrigendum No. 1 is incorporated.

11237 **NAME**

11238 fegetenv, fesetenv — get and set current floating-point environment

11239 **SYNOPSIS**

11240 #include <fenv.h>

11241 int fegetenv(fenv_t *envp);

11242 int fesetenv(const fenv_t *envp);

11243 **DESCRIPTION**

11244 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
11245 conflict between the requirements described here and the ISO C standard is unintentional. This
11246 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11247 The *fegetenv()* function shall attempt to store the current floating-point environment in the object
11248 pointed to by *envp*.

11249 The *fesetenv()* function shall attempt to establish the floating-point environment represented by
11250 the object pointed to by *envp*. The argument *envp* shall point to an object set by a call to
11251 *fegetenv()* or *feholdexcept()*, or equal a floating-point environment macro. The *fesetenv()* function
11252 does not raise floating-point exceptions, but only installs the state of the floating-point status
11253 flags represented through its argument.

11254 **RETURN VALUE**

11255 If the representation was successfully stored, *fegetenv()* shall return zero. Otherwise, it shall
11256 return a non-zero value. If the environment was successfully established, *fesetenv()* shall return
11257 zero. Otherwise, it shall return a non-zero value.

11258 **ERRORS**

11259 No errors are defined.

11260 **EXAMPLES**

11261 None.

11262 **APPLICATION USAGE**

11263 None.

11264 **RATIONALE**

11265 None.

11266 **FUTURE DIRECTIONS**

11267 None.

11268 **SEE ALSO**11269 *feholdexcept()*, *feupdateenv()*, the Base Definitions volume of IEEE Std 1003.1-2001, <fenv.h>11270 **CHANGE HISTORY**

11271 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

11272 ISO/IEC 9899:1999 standard, Technical Corrigendum No. 1 is incorporated.

11273 **NAME**

11274 fegetexceptflag, fesetexceptflag — get and set floating-point status flags

11275 **SYNOPSIS**

11276 #include <fenv.h>

11277 int fegetexceptflag(fexcept_t *flagp, int excepts);

11278 int fesetexceptflag(const fexcept_t *flagp, int excepts);

11279 **DESCRIPTION**

11280 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11281 conflict between the requirements described here and the ISO C standard is unintentional. This
11282 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11283 The *fegetexceptflag()* function shall attempt to store an implementation-defined representation of
11284 the states of the floating-point status flags indicated by the argument *excepts* in the object
11285 pointed to by the argument *flagp*.

11286 The *fesetexceptflag()* function shall attempt to set the floating-point status flags indicated by the
11287 argument *excepts* to the states stored in the object pointed to by *flagp*. The value pointed to by
11288 *flagp* shall have been set by a previous call to *fegetexceptflag()* whose second argument
11289 represented at least those floating-point exceptions represented by the argument *excepts*. This
11290 function does not raise floating-point exceptions, but only sets the state of the flags.

11291 **RETURN VALUE**

11292 If the representation was successfully stored, *fegetexceptflag()* shall return zero. Otherwise, it
11293 shall return a non-zero value. If the *excepts* argument is zero or if all the specified exceptions
11294 were successfully set, *fesetexceptflag()* shall return zero. Otherwise, it shall return a non-zero
11295 value.

11296 **ERRORS**

11297 No errors are defined.

11298 **EXAMPLES**

11299 None.

11300 **APPLICATION USAGE**

11301 None.

11302 **RATIONALE**

11303 None.

11304 **FUTURE DIRECTIONS**

11305 None.

11306 **SEE ALSO**

11307 *feclearexcept()*, *feraiseexcept()*, *fetestexcept()*, the Base Definitions volume of IEEE Std 1003.1-2001,
11308 <fenv.h>

11309 **CHANGE HISTORY**

11310 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

11311 ISO/IEC 9899:1999 standard, Technical Corrigendum No. 1 is incorporated.

11312 **NAME**

11313 fegetround, fesetround — get and set current rounding direction

11314 **SYNOPSIS**

```
11315     #include <fenv.h>

11316     int fegetround(void);
11317     int fesetround(int round);
```

11318 **DESCRIPTION**

11319 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 11320 conflict between the requirements described here and the ISO C standard is unintentional. This
 11321 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11322 The *fegetround()* function shall get the current rounding direction.

11323 The *fesetround()* function shall establish the rounding direction represented by its argument
 11324 *round*. If the argument is not equal to the value of a rounding direction macro, the rounding
 11325 direction is not changed.

11326 **RETURN VALUE**

11327 The *fegetround()* function shall return the value of the rounding direction macro representing the
 11328 current rounding direction or a negative value if there is no such rounding direction macro or
 11329 the current rounding direction is not determinable.

11330 The *fesetround()* function shall return a zero value if and only if the requested rounding direction
 11331 was established.

11332 **ERRORS**

11333 No errors are defined.

11334 **EXAMPLES**

11335 The following example saves, sets, and restores the rounding direction, reporting an error and
 11336 aborting if setting the rounding direction fails:

```
11337     #include <fenv.h>
11338     #include <assert.h>
11339     void f(int round_dir)
11340     {
11341         #pragma STDC FENV_ACCESS ON
11342         int save_round;
11343         int setround_ok;
11344         save_round = fegetround();
11345         setround_ok = fesetround(round_dir);
11346         assert(setround_ok == 0);
11347         /* ... */
11348         fesetround(save_round);
11349         /* ... */
11350     }
```

11351 **APPLICATION USAGE**

11352 None.

11353 **RATIONALE**

11354 None.

11355 **FUTURE DIRECTIONS**

11356 None.

11357 **SEE ALSO**11358 The Base Definitions volume of IEEE Std 1003.1-2001, <**fenv.h**>11359 **CHANGE HISTORY**

11360 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

11361 ISO/IEC 9899: 1999 standard, Technical Corrigendum No. 1 is incorporated.

11362 **NAME**11363 `feholdexcept` — save current floating-point environment11364 **SYNOPSIS**11365 `#include <fenv.h>`11366 `int feholdexcept(fenv_t *envp);`11367 **DESCRIPTION**

11368 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
11369 conflict between the requirements described here and the ISO C standard is unintentional. This
11370 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11371 The *feholdexcept()* function shall save the current floating-point environment in the object
11372 pointed to by *envp*, clear the floating-point status flags, and then install a non-stop (continue on
11373 floating-point exceptions) mode, if available, for all floating-point exceptions.

11374 **RETURN VALUE**

11375 The *feholdexcept()* function shall return zero if and only if non-stop floating-point exception
11376 handling was successfully installed.

11377 **ERRORS**

11378 No errors are defined.

11379 **EXAMPLES**

11380 None.

11381 **APPLICATION USAGE**

11382 None.

11383 **RATIONALE**

11384 The *feholdexcept()* function should be effective on typical IEC 60559:1989 standard
11385 implementations which have the default non-stop mode and at least one other mode for trap
11386 handling or aborting. If the implementation provides only the non-stop mode, then installing the
11387 non-stop mode is trivial.

11388 **FUTURE DIRECTIONS**

11389 None.

11390 **SEE ALSO**

11391 *fegetenv()*, *fesetenv()*, *feupdateenv()*, the Base Definitions volume of IEEE Std 1003.1-2001,
11392 `<fenv.h>`

11393 **CHANGE HISTORY**

11394 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

11395 NAME

11396 feof — test end-of-file indicator on a stream

11397 SYNOPSIS

11398 #include <stdio.h>

11399 int feof(FILE *stream);

11400 DESCRIPTION

11401 cx The functionality described on this reference page is aligned with the ISO C standard. Any
11402 conflict between the requirements described here and the ISO C standard is unintentional. This
11403 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11404 The *feof()* function shall test the end-of-file indicator for the stream pointed to by *stream*.

11405 RETURN VALUE

11406 The *feof()* function shall return non-zero if and only if the end-of-file indicator is set for *stream*.

11407 ERRORS

11408 No errors are defined.

11409 EXAMPLES

11410 None.

11411 APPLICATION USAGE

11412 None.

11413 RATIONALE

11414 None.

11415 FUTURE DIRECTIONS

11416 None.

11417 SEE ALSO

11418 *clearerr()*, *ferror()*, *fopen()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

11419 CHANGE HISTORY

11420 First released in Issue 1. Derived from Issue 1 of the SVID.

11421 **NAME**11422 `feraiseexcept` — raise floating-point exception11423 **SYNOPSIS**11424 `#include <fenv.h>`11425 `int fraiseexcept(int excepts);`11426 **DESCRIPTION**

11427 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
11428 conflict between the requirements described here and the ISO C standard is unintentional. This
11429 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11430 The `feraiseexcept()` function shall attempt to raise the supported floating-point exceptions
11431 represented by the argument *excepts*. The order in which these floating-point exceptions are
11432 raised is unspecified. Whether the `feraiseexcept()` function additionally raises the inexact
11433 floating-point exception whenever it raises the overflow or underflow floating-point exception is
11434 implementation-defined.

11435 **RETURN VALUE**

11436 If the argument is zero or if all the specified exceptions were successfully raised, `feraiseexcept()`
11437 shall return zero. Otherwise, it shall return a non-zero value.

11438 **ERRORS**

11439 No errors are defined.

11440 **EXAMPLES**

11441 None.

11442 **APPLICATION USAGE**

11443 The effect is intended to be similar to that of floating-point exceptions raised by arithmetic
11444 operations. Hence, enabled traps for floating-point exceptions raised by this function are taken.

11445 **RATIONALE**

11446 Raising overflow or underflow is allowed to also raise inexact because on some architectures the
11447 only practical way to raise an exception is to execute an instruction that has the exception as a
11448 side effect. The function is not restricted to accept only valid coincident expressions for atomic
11449 operations, so the function can be used to raise exceptions accrued over several operations.

11450 **FUTURE DIRECTIONS**

11451 None.

11452 **SEE ALSO**

11453 `feclearexcept()`, `fegetexceptflag()`, `fesetexceptflag()`, `fetestexcept()`, the Base Definitions volume of
11454 IEEE Std 1003.1-2001, `<fenv.h>`

11455 **CHANGE HISTORY**

11456 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

11457 ISO/IEC 9899:1999 standard, Technical Corrigendum No. 1 is incorporated.

11458 **NAME**

11459 error — test error indicator on a stream

11460 **SYNOPSIS**

11461 #include <stdio.h>

11462 int ferror(FILE **stream*);11463 **DESCRIPTION**

11464 cx The functionality described on this reference page is aligned with the ISO C standard. Any
11465 conflict between the requirements described here and the ISO C standard is unintentional. This
11466 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11467 The *ferror()* function shall test the error indicator for the stream pointed to by *stream*.11468 **RETURN VALUE**11469 The *ferror()* function shall return non-zero if and only if the error indicator is set for *stream*.11470 **ERRORS**

11471 No errors are defined.

11472 **EXAMPLES**

11473 None.

11474 **APPLICATION USAGE**

11475 None.

11476 **RATIONALE**

11477 None.

11478 **FUTURE DIRECTIONS**

11479 None.

11480 **SEE ALSO**11481 *clearerr()*, *feof()*, *fopen()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>11482 **CHANGE HISTORY**

11483 First released in Issue 1. Derived from Issue 1 of the SVID.

11484 **NAME**

11485 fesetenv — set current floating-point environment

11486 **SYNOPSIS**

11487 #include <fenv.h>

11488 int fesetenv(const fenv_t *envp);

11489 **DESCRIPTION**11490 Refer to *fegetenv()*.

11491 NAME

11492 `fesetexceptflag` — set floating-point status flags

11493 SYNOPSIS

11494 `#include <fenv.h>`

11495 `int fesetexceptflag(const fexcept_t *flagp, int excepts);`

11496 DESCRIPTION

11497 Refer to *fegetexceptflag()*.

11498 **NAME**

11499 fesetround — set current rounding direction

11500 **SYNOPSIS**

11501 #include <fenv.h>

11502 int fesetround(int *round*);11503 **DESCRIPTION**11504 Refer to *fegetround()*.

11505 **NAME**

11506 fetestexcept — test floating-point exception flags

11507 **SYNOPSIS**

11508 #include <fenv.h>

11509 int fetestexcept(int *excepts*);11510 **DESCRIPTION**

11511 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 11512 conflict between the requirements described here and the ISO C standard is unintentional. This
 11513 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11514 The *fetestexcept()* function shall determine which of a specified subset of the floating-point
 11515 exception flags are currently set. The *excepts* argument specifies the floating-point status flags to
 11516 be queried.

11517 **RETURN VALUE**

11518 The *fetestexcept()* function shall return the value of the bitwise-inclusive OR of the floating-point
 11519 exception macros corresponding to the currently set floating-point exceptions included in
 11520 *excepts*.

11521 **ERRORS**

11522 No errors are defined.

11523 **EXAMPLES**

11524 The following example calls function *f()* if an invalid exception is set, and then function *g()* if an
 11525 overflow exception is set:

```

11526     #include <fenv.h>
11527     /* ... */
11528     {
11529     #   pragma STDC FENV_ACCESS ON
11530         int set_excepts;
11531         feclearexcept(FE_INVALID | FE_OVERFLOW);
11532         // maybe raise exceptions
11533         set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
11534         if (set_excepts & FE_INVALID) f();
11535         if (set_excepts & FE_OVERFLOW) g();
11536         /* ... */
11537     }
```

11538 **APPLICATION USAGE**

11539 None.

11540 **RATIONALE**

11541 None.

11542 **FUTURE DIRECTIONS**

11543 None.

11544 **SEE ALSO**

11545 *feclearexcept()*, *fegetexceptflag()*, *feraiseexcept()*, the Base Definitions volume of
 11546 IEEE Std 1003.1-2001, <fenv.h>

11547 **CHANGE HISTORY**

11548 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

11549 **NAME**

11550 feupdateenv — update floating-point environment

11551 **SYNOPSIS**

11552 #include <fenv.h>

11553 int feupdateenv(const fenv_t *envp);

11554 **DESCRIPTION**

11555 cx The functionality described on this reference page is aligned with the ISO C standard. Any
11556 conflict between the requirements described here and the ISO C standard is unintentional. This
11557 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11558 The *feupdateenv()* function shall attempt to save the currently raised floating-point exceptions in
11559 its automatic storage, attempt to install the floating-point environment represented by the object
11560 pointed to by *envp*, and then attempt to raise the saved floating-point exceptions. The argument
11561 *envp* shall point to an object set by a call to *feholdexcept()* or *fegetenv()*, or equal a floating-point
11562 environment macro.

11563 **RETURN VALUE**

11564 The *feupdateenv()* function shall return a zero value if and only if all the required actions were
11565 successfully carried out.

11566 **ERRORS**

11567 No errors are defined.

11568 **EXAMPLES**

11569 The following example shows sample code to hide spurious underflow floating-point
11570 exceptions:

```
11571 #include <fenv.h>
11572 double f(double x)
11573 {
11574     # pragma STDC FENV_ACCESS ON
11575     double result;
11576     fenv_t save_env;
11577     feholdexcept(&save_env);
11578     // compute result
11579     if (/* test spurious underflow */)
11580         feclearexcept(FE_UNDERFLOW);
11581     feupdateenv(&save_env);
11582     return result;
11583 }
```

11584 **APPLICATION USAGE**

11585 None.

11586 **RATIONALE**

11587 None.

11588 **FUTURE DIRECTIONS**

11589 None.

11590 **SEE ALSO**11591 *fegetenv()*, *feholdexcept()*, the Base Definitions volume of IEEE Std 1003.1-2001, <fenv.h>

CHANGE HISTORY

- 11592
- 11593 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.
- 11594 ISO/IEC 9899: 1999 standard, Technical Corrigendum No. 1 is incorporated.

11595 **NAME**

11596 fflush — flush a stream

11597 **SYNOPSIS**

11598 #include <stdio.h>

11599 int fflush(FILE **stream*);11600 **DESCRIPTION**

11601 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 11602 conflict between the requirements described here and the ISO C standard is unintentional. This
 11603 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11604 If *stream* points to an output stream or an update stream in which the most recent operation was
 11605 CX not input, *fflush()* shall cause any unwritten data for that stream to be written to the file, and the
 11606 *st_ctime* and *st_mtime* fields of the underlying file shall be marked for update.

11607 If *stream* is a null pointer, *fflush()* shall perform this flushing action on all streams for which the
 11608 behavior is defined above.

11609 **RETURN VALUE**

11610 Upon successful completion, *fflush()* shall return 0; otherwise, it shall set the error indicator for
 11611 CX the stream, return EOF, and set *errno* to indicate the error.

11612 **ERRORS**11613 The *fflush()* function shall fail if:

11614 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the
 11615 process would be delayed in the write operation.

11616 CX [EBADF] The file descriptor underlying *stream* is not valid.

11617 CX [EFBIG] An attempt was made to write a file that exceeds the maximum file size.

11618 XSI [EFBIG] An attempt was made to write a file that exceeds the process' file size limit.

11619 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
 11620 offset maximum associated with the corresponding stream.

11621 CX [EINTR] The *fflush()* function was interrupted by a signal.

11622 CX [EIO] The process is a member of a background process group attempting to write
 11623 to its controlling terminal, TOSTOP is set, the process is neither ignoring nor
 11624 blocking SIGTTOU, and the process group of the process is orphaned. This
 11625 error may also be returned under implementation-defined conditions.

11626 CX [ENOSPC] There was no free space remaining on the device containing the file.

11627 CX [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by
 11628 any process. A SIGPIPE signal shall also be sent to the thread.

11629 The *fflush()* function may fail if:

11630 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
 11631 capabilities of the device.

11632 **EXAMPLES**11633 **Sending Prompts to Standard Output**

11634 The following example uses *printf()* calls to print a series of prompts for information the user
11635 must enter from standard input. The *fflush()* calls force the output to standard output. The
11636 *fflush()* function is used because standard output is usually buffered and the prompt may not
11637 immediately be printed on the output or terminal. The *gets()* calls read strings from standard
11638 input and place the results in variables, for use later in the program.

```
11639 #include <stdio.h>
11640 ...
11641 char user[100];
11642 char oldpasswd[100];
11643 char newpasswd[100];
11644 ...
11645 printf("User name: ");
11646 fflush(stdout);
11647 gets(user);

11648 printf("Old password: ");
11649 fflush(stdout);
11650 gets(oldpasswd);

11651 printf("New password: ");
11652 fflush(stdout);
11653 gets(newpasswd);
11654 ...
```

11655 **APPLICATION USAGE**

11656 None.

11657 **RATIONALE**

11658 Data buffered by the system may make determining the validity of the position of the current
11659 file descriptor impractical. Thus, enforcing the repositioning of the file descriptor after *fflush()*
11660 on streams open for *read()* is not mandated by IEEE Std 1003.1-2001.

11661 **FUTURE DIRECTIONS**

11662 None.

11663 **SEE ALSO**

11664 *getrlimit()*, *ulimit()*, the Base Definitions volume of IEEE Std 1003.1-2001, *<stdio.h>*

11665 **CHANGE HISTORY**

11666 First released in Issue 1. Derived from Issue 1 of the SVID.

11667 **Issue 5**

11668 Large File Summit extensions are added.

11669 **Issue 6**

11670 Extensions beyond the ISO C standard are marked.

11671 The following new requirements on POSIX implementations derive from alignment with the
11672 Single UNIX Specification:

- 11673 • The [EFBIG] error is added as part of the large file support extensions.
- 11674 • The [ENXIO] optional error condition is added.

11675 The RETURN VALUE section is updated to note that the error indicator shall be set for the
11676 stream. This is for alignment with the ISO/IEC 9899:1999 standard.

11677 NAME

11678 ffs — find first set bit

11679 SYNOPSIS

11680 xSI #include <strings.h>

11681 int ffs(int i);

11682

11683 DESCRIPTION

11684 The *ffs()* function shall find the first bit set (beginning with the least significant bit) in *i*, and
11685 return the index of that bit. Bits are numbered starting at one (the least significant bit).

11686 RETURN VALUE

11687 The *ffs()* function shall return the index of the first bit set. If *i* is 0, then *ffs()* shall return 0.

11688 ERRORS

11689 No errors are defined.

11690 EXAMPLES

11691 None.

11692 APPLICATION USAGE

11693 None.

11694 RATIONALE

11695 None.

11696 FUTURE DIRECTIONS

11697 None.

11698 SEE ALSO

11699 The Base Definitions volume of IEEE Std 1003.1-2001, <**strings.h**>

11700 CHANGE HISTORY

11701 First released in Issue 4, Version 2.

11702 Issue 5

11703 Moved from X/OPEN UNIX extension to BASE.

11704 NAME

11705 fgetc — get a byte from a stream

11706 SYNOPSIS

11707 #include <stdio.h>

11708 int fgetc(FILE *stream);

11709 DESCRIPTION

11710 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 11711 conflict between the requirements described here and the ISO C standard is unintentional. This
 11712 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11713 If the end-of-file indicator for the input stream pointed to by *stream* is not set and a next byte is
 11714 present, the *fgetc()* function shall obtain the next byte as an **unsigned char** converted to an **int**,
 11715 from the input stream pointed to by *stream*, and advance the associated file position indicator for
 11716 the stream (if defined). Since *fgetc()* operates on bytes, reading a character consisting of multiple
 11717 bytes (or “a multi-byte character”) may require multiple calls to *fgetc()*.

11718 CX The *fgetc()* function may mark the *st_atime* field of the file associated with *stream* for update. The
 11719 *st_atime* field shall be marked for update by the first successful execution of *fgetc()*, *fgets()*,
 11720 *fgetwc()*, *fgetws()*, *fread()*, *fscanf()*, *getc()*, *getchar()*, *gets()*, or *scanf()* using *stream* that returns
 11721 data not supplied by a prior call to *ungetc()* or *ungetwc()*.

11722 RETURN VALUE

11723 Upon successful completion, *fgetc()* shall return the next byte from the input stream pointed to
 11724 by *stream*. If the end-of-file indicator for the stream is set, or if the stream is at end-of-file, the
 11725 end-of-file indicator for the stream shall be set and *fgetc()* shall return EOF. If a read error occurs,
 11726 CX the error indicator for the stream shall be set, *fgetc()* shall return EOF, and shall set *errno* to
 11727 indicate the error.

11728 ERRORS

11729 The *fgetc()* function shall fail if data needs to be read and:

11730 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the
 11731 process would be delayed in the *fgetc()* operation.

11732 CX [EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for
 11733 reading.

11734 CX [EINTR] The read operation was terminated due to the receipt of a signal, and no data
 11735 was transferred.

11736 CX [EIO] A physical I/O error has occurred, or the process is in a background process
 11737 group attempting to read from its controlling terminal, and either the process
 11738 is ignoring or blocking the SIGTIN signal or the process group is orphaned.
 11739 This error may also be generated for implementation-defined reasons.

11740 CX [EOVERFLOW] The file is a regular file and an attempt was made to read at or beyond the
 11741 offset maximum associated with the corresponding stream.

11742 The *fgetc()* function may fail if:

11743 CX [ENOMEM] Insufficient storage space is available.

11744 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
 11745 capabilities of the device.

11746 **EXAMPLES**

11747 None.

11748 **APPLICATION USAGE**

11749 If the integer value returned by *fgetc()* is stored into a variable of type **char** and then compared
11750 against the integer constant EOF, the comparison may never succeed, because sign-extension of
11751 a variable of type **char** on widening to integer is implementation-defined.

11752 The *ferror()* or *feof()* functions must be used to distinguish between an error condition and an
11753 end-of-file condition.

11754 **RATIONALE**

11755 None.

11756 **FUTURE DIRECTIONS**

11757 None.

11758 **SEE ALSO**

11759 *feof()*, *ferror()*, *fopen()*, *getchar()*, *getc()*, the Base Definitions volume of IEEE Std 1003.1-2001,
11760 <stdio.h>

11761 **CHANGE HISTORY**

11762 First released in Issue 1. Derived from Issue 1 of the SVID.

11763 **Issue 5**

11764 Large File Summit extensions are added.

11765 **Issue 6**

11766 Extensions beyond the ISO C standard are marked.

11767 The following new requirements on POSIX implementations derive from alignment with the
11768 Single UNIX Specification:

- 11769 • The [EIO] and [EOVERFLOW] mandatory error conditions are added.
- 11770 • The [ENOMEM] and [ENXIO] optional error conditions are added.

11771 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 11772 • The DESCRIPTION is updated to clarify the behavior when the end-of-file indicator for the
11773 input stream is not set.
- 11774 • The RETURN VALUE section is updated to note that the error indicator shall be set for the
11775 stream.

11776 **NAME**

11777 fgetpos — get current file position information

11778 **SYNOPSIS**

11779 #include <stdio.h>

11780 int fgetpos(FILE *restrict stream, fpos_t *restrict pos);

11781 **DESCRIPTION**

11782 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 11783 conflict between the requirements described here and the ISO C standard is unintentional. This
 11784 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11785 The *fgetpos()* function shall store the current values of the parse state (if any) and file position
 11786 indicator for the stream pointed to by *stream* in the object pointed to by *pos*. The value stored
 11787 contains unspecified information usable by *fsetpos()* for repositioning the stream to its position
 11788 at the time of the call to *fgetpos()*.

11789 **RETURN VALUE**

11790 Upon successful completion, *fgetpos()* shall return 0; otherwise, it shall return a non-zero value
 11791 and set *errno* to indicate the error.

11792 **ERRORS**11793 The *fgetpos()* function shall fail if:

11794 CX [EOVERFLOW] The current value of the file position cannot be represented correctly in an
 11795 object of type **fpos_t**.

11796 The *fgetpos()* function may fail if:

11797 CX [EBADF] The file descriptor underlying *stream* is not valid.

11798 CX [ESPIPE] The file descriptor underlying *stream* is associated with a pipe, FIFO, or socket.
 11799

11800 **EXAMPLES**

11801 None.

11802 **APPLICATION USAGE**

11803 None.

11804 **RATIONALE**

11805 None.

11806 **FUTURE DIRECTIONS**

11807 None.

11808 **SEE ALSO**11809 *fopen()*, *ftell()*, *rewind()*, *ungetc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>11810 **CHANGE HISTORY**

11811 First released in Issue 4. Derived from the ISO C standard.

11812 **Issue 5**

11813 Large File Summit extensions are added.

11814 **Issue 6**

11815 Extensions beyond the ISO C standard are marked.

11816 The following new requirements on POSIX implementations derive from alignment with the
 11817 Single UNIX Specification:

- 11818 • The [EBADF] and [ESPIPE] optional error conditions are added.
- 11819 An additional [ESPIPE] error condition is added for sockets.
- 11820 The prototype for *fgetpos()* is changed for alignment with the ISO/IEC 9899:1999 standard.

11821 **NAME**

11822 fgets — get a string from a stream

11823 **SYNOPSIS**

11824 #include <stdio.h>

11825 char *fgets(char *restrict *s*, int *n*, FILE *restrict *stream*);11826 **DESCRIPTION**

11827 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 11828 conflict between the requirements described here and the ISO C standard is unintentional. This
 11829 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11830 The *fgets()* function shall read bytes from *stream* into the array pointed to by *s*, until *n*−1 bytes
 11831 are read, or a <newline> is read and transferred to *s*, or an end-of-file condition is encountered.
 11832 The string is then terminated with a null byte.

11833 CX The *fgets()* function may mark the *st_atime* field of the file associated with *stream* for update. The
 11834 *st_atime* field shall be marked for update by the first successful execution of *fgetc()*, *fgets()*,
 11835 *fgetwc()*, *fgetws()*, *fread()*, *fscanf()*, *getc()*, *getchar()*, *gets()*, or *scanf()* using *stream* that returns
 11836 data not supplied by a prior call to *ungetc()* or *ungetwc()*.

11837 **RETURN VALUE**

11838 Upon successful completion, *fgets()* shall return *s*. If the stream is at end-of-file, the end-of-file
 11839 indicator for the stream shall be set and *fgets()* shall return a null pointer. If a read error occurs,
 11840 CX the error indicator for the stream shall be set, *fgets()* shall return a null pointer, and shall set
 11841 *errno* to indicate the error.

11842 **ERRORS**11843 Refer to *fgetc()*.11844 **EXAMPLES**11845 **Reading Input**

11846 The following example uses *fgets()* to read each line of input. {LINE_MAX}, which defines the
 11847 maximum size of the input line, is defined in the <limits.h> header.

```

11848       #include <stdio.h>
11849       ...
11850       char line[LINE_MAX];
11851       ...
11852       while (fgets(line, LINE_MAX, fp) != NULL) {
11853       ...
11854       }
11855       ...
```

11856 **APPLICATION USAGE**

11857 None.

11858 **RATIONALE**

11859 None.

11860 **FUTURE DIRECTIONS**

11861 None.

11862 **SEE ALSO**

11863 *fopen()*, *fread()*, *gets()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

11864 **CHANGE HISTORY**

11865 First released in Issue 1. Derived from Issue 1 of the SVID.

11866 **Issue 6**

11867 Extensions beyond the ISO C standard are marked.

11868 The prototype for *fgets()* is changed for alignment with the ISO/IEC 9899:1999 standard.

11869 **NAME**

11870 fgetwc — get a wide-character code from a stream

11871 **SYNOPSIS**

11872 #include <stdio.h>

11873 #include <wchar.h>

11874 wint_t fgetwc(FILE *stream);

11875 **DESCRIPTION**

11876 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 11877 conflict between the requirements described here and the ISO C standard is unintentional. This
 11878 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11879 The *fgetwc()* function shall obtain the next character (if present) from the input stream pointed to
 11880 by *stream*, convert that to the corresponding wide-character code, and advance the associated
 11881 file position indicator for the stream (if defined).

11882 If an error occurs, the resulting value of the file position indicator for the stream is unspecified.

11883 CX The *fgetwc()* function may mark the *st_atime* field of the file associated with *stream* for update.
 11884 The *st_atime* field shall be marked for update by the first successful execution of *fgetc()*, *fgets()*,
 11885 *fgetwc()*, *fgetws()*, *fread()*, *fscanf()*, *getc()*, *getchar()*, *gets()*, or *scanf()* using *stream* that returns
 11886 data not supplied by a prior call to *ungetc()* or *ungetwc()*.

11887 **RETURN VALUE**

11888 Upon successful completion, the *fgetwc()* function shall return the wide-character code of the
 11889 character read from the input stream pointed to by *stream* converted to a type **wint_t**. If the
 11890 stream is at end-of-file, the end-of-file indicator for the stream shall be set and *fgetwc()* shall
 11891 return WEOF. If a read error occurs, the error indicator for the stream shall be set, *fgetwc()* shall
 11892 CX return WEOF, and shall set *errno* to indicate the error. If an encoding error occurs, the error
 11893 indicator for the stream shall be set, *fgetwc()* shall return WEOF, and shall set *errno* to indicate
 11894 the error.

11895 **ERRORS**

11896 The *fgetwc()* function shall fail if data needs to be read and:

11897 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the
 11898 process would be delayed in the *fgetwc()* operation.

11899 CX [EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for
 11900 reading.

11901 [EILSEQ] The data obtained from the input stream does not form a valid character.

11902 CX [EINTR] The read operation was terminated due to the receipt of a signal, and no data
 11903 was transferred.

11904 CX [EIO] A physical I/O error has occurred, or the process is in a background process
 11905 group attempting to read from its controlling terminal, and either the process
 11906 is ignoring or blocking the SIGTTIN signal or the process group is orphaned.
 11907 This error may also be generated for implementation-defined reasons.

11908 CX [EOVERFLOW] The file is a regular file and an attempt was made to read at or beyond the
 11909 offset maximum associated with the corresponding stream.

11910 The *fgetwc()* function may fail if:

11911 CX [ENOMEM] Insufficient storage space is available.

11912 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
 11913 capabilities of the device.

11914 EXAMPLES

11915 None.

11916 APPLICATION USAGE

11917 The *ferror()* or *feof()* functions must be used to distinguish between an error condition and an
 11918 end-of-file condition.

11919 RATIONALE

11920 None.

11921 FUTURE DIRECTIONS

11922 None.

11923 SEE ALSO

11924 *feof()*, *ferror()*, *fopen()*, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdio.h>`,
 11925 `<wchar.h>`

11926 CHANGE HISTORY

11927 First released in Issue 4. Derived from the MSE working draft.

11928 Issue 5

11929 The Optional Header (OH) marking is removed from `<stdio.h>`.

11930 Large File Summit extensions are added.

11931 Issue 6

11932 Extensions beyond the ISO C standard are marked.

11933 The following new requirements on POSIX implementations derive from alignment with the
 11934 Single UNIX Specification:

- 11935 • The [EIO] and [EOVERFLOW] mandatory error conditions are added.
- 11936 • The [ENOMEM] and [ENXIO] optional error conditions are added.

11937 **NAME**11938 `fgetws` — get a wide-character string from a stream11939 **SYNOPSIS**11940 `#include <stdio.h>`11941 `#include <wchar.h>`11942 `wchar_t *fgetws(wchar_t *restrict ws, int n,`11943 `FILE *restrict stream);`11944 **DESCRIPTION**

11945 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 11946 conflict between the requirements described here and the ISO C standard is unintentional. This
 11947 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11948 The `fgetws()` function shall read characters from the *stream*, convert these to the corresponding
 11949 wide-character codes, place them in the `wchar_t` array pointed to by *ws*, until *n*−1 characters are
 11950 read, or a <newline> is read, converted, and transferred to *ws*, or an end-of-file condition is
 11951 encountered. The wide-character string, *ws*, shall then be terminated with a null wide-character
 11952 code.

11953 If an error occurs, the resulting value of the file position indicator for the stream is unspecified.

11954 CX The `fgetws()` function may mark the *st_atime* field of the file associated with *stream* for update.
 11955 The *st_atime* field shall be marked for update by the first successful execution of `fgetc()`, `fgets()`,
 11956 `fgetwc()`, `fgetws()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `gets()`, or `scanf()` using *stream* that returns
 11957 data not supplied by a prior call to `ungetc()` or `ungetwc()`.

11958 **RETURN VALUE**

11959 Upon successful completion, `fgetws()` shall return *ws*. If the stream is at end-of-file, the end-of-
 11960 file indicator for the stream shall be set and `fgetws()` shall return a null pointer. If a read error
 11961 CX occurs, the error indicator for the stream shall be set, `fgetws()` shall return a null pointer, and
 11962 shall set *errno* to indicate the error.

11963 **ERRORS**11964 Refer to `fgetwc()`.11965 **EXAMPLES**

11966 None.

11967 **APPLICATION USAGE**

11968 None.

11969 **RATIONALE**

11970 None.

11971 **FUTURE DIRECTIONS**

11972 None.

11973 **SEE ALSO**11974 `fopen()`, `fread()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdio.h>`, `<wchar.h>`11975 **CHANGE HISTORY**

11976 First released in Issue 4. Derived from the MSE working draft.

11977 **Issue 5**11978 The Optional Header (OH) marking is removed from `<stdio.h>`.

11979 **Issue 6**

11980 Extensions beyond the ISO C standard are marked.

11981 The prototype for *fgetws()* is changed for alignment with the ISO/IEC 9899:1999 standard.

11982 NAME

11983 `fileno` — map a stream pointer to a file descriptor

11984 SYNOPSIS

11985 `cx` `#include <stdio.h>`

11986 `int fileno(FILE *stream);`

11987

11988 DESCRIPTION

11989 The `fileno()` function shall return the integer file descriptor associated with the stream pointed to
11990 by `stream`.

11991 RETURN VALUE

11992 Upon successful completion, `fileno()` shall return the integer value of the file descriptor
11993 associated with `stream`. Otherwise, the value `-1` shall be returned and `errno` set to indicate the
11994 error.

11995 ERRORS

11996 The `fileno()` function may fail if:

11997 [EBADF] The `stream` argument is not a valid stream.

11998 EXAMPLES

11999 None.

12000 APPLICATION USAGE

12001 None.

12002 RATIONALE

12003 Without some specification of which file descriptors are associated with these streams, it is
12004 impossible for an application to set up the streams for another application it starts with `fork()`
12005 and `exec`. In particular, it would not be possible to write a portable version of the `sh` command
12006 interpreter (although there may be other constraints that would prevent that portability).

12007 FUTURE DIRECTIONS

12008 None.

12009 SEE ALSO

12010 Section 2.5.1 (on page 35), `fdopen()`, `fopen()`, `stdin`, the Base Definitions volume of
12011 IEEE Std 1003.1-2001, `<stdio.h>`

12012 CHANGE HISTORY

12013 First released in Issue 1. Derived from Issue 1 of the SVID.

12014 Issue 6

12015 The following new requirements on POSIX implementations derive from alignment with the
12016 Single UNIX Specification:

- 12017 • The [EBADF] optional error condition is added.

12018 **NAME**

12019 flockfile, ftrylockfile, funlockfile — stdio locking functions

12020 **SYNOPSIS**12021 TSF `#include <stdio.h>`

```
12022 void flockfile(FILE *file);
12023 int ftrylockfile(FILE *file);
12024 void funlockfile(FILE *file);
12025
```

12026 **DESCRIPTION**

12027 These functions shall provide for explicit application-level locking of stdio (**FILE** *) objects.
 12028 These functions can be used by a thread to delineate a sequence of I/O statements that are
 12029 executed as a unit.

12030 The *flockfile()* function shall acquire for a thread ownership of a (**FILE** *) object.

12031 The *ftrylockfile()* function shall acquire for a thread ownership of a (**FILE** *) object if the object is
 12032 available; *ftrylockfile()* is a non-blocking version of *flockfile()*.

12033 The *funlockfile()* function shall relinquish the ownership granted to the thread. The behavior is
 12034 undefined if a thread other than the current owner calls the *funlockfile()* function.

12035 The functions shall behave as if there is a lock count associated with each (**FILE** *) object. This
 12036 count is implicitly initialized to zero when the (**FILE** *) object is created. The (**FILE** *) object is
 12037 unlocked when the count is zero. When the count is positive, a single thread owns the (**FILE** *)
 12038 object. When the *flockfile()* function is called, if the count is zero or if the count is positive and
 12039 the caller owns the (**FILE** *) object, the count shall be incremented. Otherwise, the calling thread
 12040 shall be suspended, waiting for the count to return to zero. Each call to *funlockfile()* shall
 12041 decrement the count. This allows matching calls to *flockfile()* (or successful calls to *ftrylockfile()*)
 12042 and *funlockfile()* to be nested.

12043 All functions that reference (**FILE** *) objects shall behave as if they use *flockfile()* and *funlockfile()*
 12044 internally to obtain ownership of these (**FILE** *) objects.

12045 **RETURN VALUE**

12046 None for *flockfile()* and *funlockfile()*.

12047 The *ftrylockfile()* function shall return zero for success and non-zero to indicate that the lock
 12048 cannot be acquired.

12049 **ERRORS**

12050 No errors are defined.

12051 **EXAMPLES**

12052 None.

12053 **APPLICATION USAGE**

12054 Applications using these functions may be subject to priority inversion, as discussed in the Base
 12055 Definitions volume of IEEE Std 1003.1-2001, Section 3.285, Priority Inversion.

12056 **RATIONALE**

12057 The *flockfile()* and *funlockfile()* functions provide an orthogonal mutual-exclusion lock for each
 12058 **FILE**. The *ftrylockfile()* function provides a non-blocking attempt to acquire a file lock,
 12059 analogous to *pthread_mutex_trylock()*.

12060 These locks behave as if they are the same as those used internally by *stdio* for thread-safety.
 12061 This both provides thread-safety of these functions without requiring a second level of internal
 12062 locking and allows functions in *stdio* to be implemented in terms of other *stdio* functions.

12063 Application writers and implementors should be aware that there are potential deadlock
12064 problems on **FILE** objects. For example, the line-buffered flushing semantics of *stdio* (requested
12065 via `{_IOLBF}`) require that certain input operations sometimes cause the buffered contents of
12066 implementation-defined line-buffered output streams to be flushed. If two threads each hold the
12067 lock on the other's **FILE**, deadlock ensues. This type of deadlock can be avoided by acquiring
12068 **FILE** locks in a consistent order. In particular, the line-buffered output stream deadlock can
12069 typically be avoided by acquiring locks on input streams before locks on output streams if a
12070 thread would be acquiring both.

12071 In summary, threads sharing *stdio* streams with other threads can use *flockfile()* and *funlockfile()*
12072 to cause sequences of I/O performed by a single thread to be kept bundled. The only case where
12073 the use of *flockfile()* and *funlockfile()* is required is to provide a scope protecting uses of the
12074 `*_unlocked()` functions/macros. This moves the cost/performance tradeoff to the optimal point.

12075 FUTURE DIRECTIONS

12076 None.

12077 SEE ALSO

12078 *getc_unlocked()*, *putc_unlocked()*, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdio.h>`

12079 CHANGE HISTORY

12080 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

12081 Issue 6

12082 These functions are marked as part of the Thread-Safe Functions option.

12083 NAME

12084 floor, floorf, floorl — floor function

12085 SYNOPSIS

12086 #include <math.h>

12087 double floor(double x);

12088 float floorf(float x);

12089 long double floorl(long double x);

12090 DESCRIPTION

12091 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 12092 conflict between the requirements described here and the ISO C standard is unintentional. This
 12093 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

12094 These functions shall compute the largest integral value not greater than *x*.

12095 An application wishing to check for error situations should set *errno* to zero and call
 12096 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 12097 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 12098 zero, an error has occurred.

12099 RETURN VALUE

12100 Upon successful completion, these functions shall return the largest integral value not greater
 12101 than *x*, expressed as a **double**, **float**, or **long double**, as appropriate for the return type of the
 12102 function.

12103 MX If *x* is NaN, a NaN shall be returned.12104 If *x* is ± 0 or $\pm \text{Inf}$, *x* shall be returned.

12105 XSI If the correct value would cause overflow, a range error shall occur and *floor*(), *floorf*(), and
 12106 *floorl*() shall return the value of the macro `-HUGE_VAL`, `-HUGE_VALF`, and `-HUGE_VALL`,
 12107 respectively.

12108 ERRORS

12109 These functions shall fail if:

12110 XSI Range Error The result would cause an overflow.

12111 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 12112 then *errno* shall be set to [ERANGE]. If the integer expression
 12113 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow
 12114 floating-point exception shall be raised.

12115 EXAMPLES

12116 None.

12117 APPLICATION USAGE

12118 The integral value returned by these functions might not be expressible as an **int** or **long**. The
 12119 return value should be tested before assigning it to an integer type to avoid the undefined results
 12120 of an integer overflow.

12121 The *floor*() function can only overflow when the floating-point representation has
 12122 `DBL_MANT_DIG > DBL_MAX_EXP`.

12123 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
 12124 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

12125 **RATIONALE**

12126 None.

12127 **FUTURE DIRECTIONS**

12128 None.

12129 **SEE ALSO**

12130 *ceil()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-2001,
12131 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

12132 **CHANGE HISTORY**

12133 First released in Issue 1. Derived from Issue 1 of the SVID.

12134 **Issue 5**

12135 The DESCRIPTION is updated to indicate how an application should check for an error. This
12136 text was previously published in the APPLICATION USAGE section.

12137 **Issue 6**12138 The *floorf()* and *floorl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

12139 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
12140 revised to align with the ISO/IEC 9899:1999 standard.

12141 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
12142 marked.

12143 **NAME**

12144 fma, fmaf, fmal — floating-point multiply-add

12145 **SYNOPSIS**

12146 #include <math.h>

12147 double fma(double x, double y, double z);

12148 float fmaf(float x, float y, float z);

12149 long double fmal(long double x, long double y, long double z);

12150 **DESCRIPTION**

12151 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 12152 conflict between the requirements described here and the ISO C standard is unintentional. This
 12153 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

12154 These functions shall compute $(x * y) + z$, rounded as one ternary operation: they shall compute
 12155 the value (as if) to infinite precision and round once to the result format, according to the
 12156 rounding mode characterized by the value of FLT_ROUNDS.

12157 An application wishing to check for error situations should set *errno* to zero and call
 12158 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 12159 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 12160 zero, an error has occurred.

12161 **RETURN VALUE**

12162 Upon successful completion, these functions shall return $(x * y) + z$, rounded as one ternary
 12163 operation.

12164 **MX** If *x* or *y* are NaN, a NaN shall be returned.

12165 If *x* multiplied by *y* is an exact infinity and *z* is also an infinity but with the opposite sign, a
 12166 domain error shall occur, and either a NaN (if supported), or an implementation-defined value
 12167 shall be returned.

12168 If one of *x* and *y* is infinite, the other is zero, and *z* is not a NaN, a domain error shall occur, and
 12169 either a NaN (if supported), or an implementation-defined value shall be returned.

12170 If one of *x* and *y* is infinite, the other is zero, and *z* is a NaN, a NaN shall be returned and a
 12171 domain error may occur.

12172 If $x * y$ is not $0 * \text{Inf}$ nor $\text{Inf} * 0$ and *z* is a NaN, a NaN shall be returned.

12173 **ERRORS**

12174 These functions shall fail if:

12175 **MX** **Domain Error** The value of $x * y + z$ is invalid, or the value $x * y$ is invalid and *z* is not a NaN.

12176 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 12177 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling
 12178 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 12179 shall be raised.

12180 **MX** **Range Error** The result overflows.

12181 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 12182 then *errno* shall be set to [ERANGE]. If the integer expression
 12183 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow
 12184 floating-point exception shall be raised.

12185 These functions may fail if:

12186 MX **Domain Error** The value $x*y$ is invalid and z is a NaN.

12187 If the integer expression `(math_errhandling & MATH_ERRNO)` is non-zero,
 12188 then *errno* shall be set to [EDOM]. If the integer expression `(math_errhandling`
 12189 `& MATH_ERREXCEPT)` is non-zero, then the invalid floating-point exception
 12190 shall be raised.

12191 MX **Range Error** The result underflows.

12192 If the integer expression `(math_errhandling & MATH_ERRNO)` is non-zero,
 12193 then *errno* shall be set to [ERANGE]. If the integer expression
 12194 `(math_errhandling & MATH_ERREXCEPT)` is non-zero, then the underflow
 12195 floating-point exception shall be raised.

12196 EXAMPLES

12197 None.

12198 APPLICATION USAGE

12199 On error, the expressions `(math_errhandling & MATH_ERRNO)` and `(math_errhandling &`
 12200 `MATH_ERREXCEPT)` are independent of each other, but at least one of them must be non-zero.

12201 RATIONALE

12202 In many cases, clever use of floating (*fused*) multiply-add leads to much improved code; but its
 12203 unexpected use by the compiler can undermine carefully written code. The FP_CONTRACT
 12204 macro can be used to disallow use of floating multiply-add; and the *fma()* function guarantees
 12205 its use where desired. Many current machines provide hardware floating multiply-add
 12206 instructions; software implementation can be used for others.

12207 FUTURE DIRECTIONS

12208 None.

12209 SEE ALSO

12210 *feclearexcept()*, *fetestexcept()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18,
 12211 Treatment of Error Conditions for Mathematical Functions, <math.h>

12212 CHANGE HISTORY

12213 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

12214 **NAME**

12215 fmax, fmaxf, fmaxl — determine maximum numeric value of two floating-point numbers

12216 **SYNOPSIS**

12217 #include <math.h>

12218 double fmax(double x, double y);

12219 float fmaxf(float x, float y);

12220 long double fmaxl(long double x, long double y);

12221 **DESCRIPTION**

12222 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12223 conflict between the requirements described here and the ISO C standard is unintentional. This
12224 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

12225 These functions shall determine the maximum numeric value of their arguments. NaN
12226 arguments shall be treated as missing data: if one argument is a NaN and the other numeric,
12227 then these functions shall choose the numeric value.

12228 **RETURN VALUE**

12229 Upon successful completion, these functions shall return the maximum numeric value of their
12230 arguments.

12231 If just one argument is a NaN, the other argument shall be returned.

12232 MX If *x* and *y* are NaN, a NaN shall be returned.

12233 **ERRORS**

12234 No errors are defined.

12235 **EXAMPLES**

12236 None.

12237 **APPLICATION USAGE**

12238 None.

12239 **RATIONALE**

12240 None.

12241 **FUTURE DIRECTIONS**

12242 None.

12243 **SEE ALSO**

12244 *fdim()*, *fmin()*, the Base Definitions volume of IEEE Std 1003.1-2001, <math.h>

12245 **CHANGE HISTORY**

12246 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

12247 **NAME**

12248 fmin, fminf, fminl — determine minimum numeric value of two floating-point numbers

12249 **SYNOPSIS**

12250 #include <math.h>

12251 double fmin(double x, double y);

12252 float fminf(float x, float y);

12253 long double fminl(long double x, long double y);

12254 **DESCRIPTION**

12255 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
12256 conflict between the requirements described here and the ISO C standard is unintentional. This
12257 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

12258 These functions shall determine the minimum numeric value of their arguments. NaN
12259 arguments shall be treated as missing data: if one argument is a NaN and the other numeric,
12260 then these functions shall choose the numeric value.

12261 **RETURN VALUE**

12262 Upon successful completion, these functions shall return the minimum numeric value of their
12263 arguments.

12264 If just one argument is a NaN, the other argument shall be returned.

12265 **MX** If *x* and *y* are NaN, a NaN shall be returned.

12266 **ERRORS**

12267 No errors are defined.

12268 **EXAMPLES**

12269 None.

12270 **APPLICATION USAGE**

12271 None.

12272 **RATIONALE**

12273 None.

12274 **FUTURE DIRECTIONS**

12275 None.

12276 **SEE ALSO**

12277 *fdim()*, *fmax()*, the Base Definitions volume of IEEE Std 1003.1-2001, <math.h>

12278 **CHANGE HISTORY**

12279 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

12280 **NAME**

12281 fmod, fmodf, fmodl — floating-point remainder value function

12282 **SYNOPSIS**

12283 #include <math.h>

12284 double fmod(double x, double y);

12285 float fmodf(float x, float y);

12286 long double fmodl(long double x, long double y);

12287 **DESCRIPTION**

12288 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 12289 conflict between the requirements described here and the ISO C standard is unintentional. This
 12290 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

12291 These functions shall return the floating-point remainder of the division of *x* by *y*.

12292 An application wishing to check for error situations should set *errno* to zero and call
 12293 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 12294 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 12295 zero, an error has occurred.

12296 **RETURN VALUE**

12297 These functions shall return the value $x - i * y$, for some integer *i* such that, if *y* is non-zero, the
 12298 result has the same sign as *x* and magnitude less than the magnitude of *y*.

12299 If the correct value would cause underflow, and is not representable, a range error may occur,
 12300 MX and either 0.0 (if supported), or an implementation-defined value shall be returned.

12301 MX If *x* or *y* is NaN, a NaN shall be returned.

12302 If *y* is zero, a domain error shall occur, and either a NaN (if supported), or an implementation-
 12303 defined value shall be returned.

12304 If *x* is infinite, a domain error shall occur, and either a NaN (if supported), or an
 12305 implementation-defined value shall be returned.

12306 If *x* is ± 0 and *y* is not zero, ± 0 shall be returned.12307 If *x* is not infinite and *y* is $\pm \text{Inf}$, *x* shall be returned.

12308 If the correct value would cause underflow, and is representable, a range error may occur and
 12309 the correct value shall be returned.

12310 **ERRORS**

12311 These functions shall fail if:

12312 MX **Domain Error** The *x* argument is infinite or *y* is zero.

12313 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 12314 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling
 12315 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 12316 shall be raised.

12317 These functions may fail if:

12318 **Range Error** The result underflows.

12319 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 12320 then *errno* shall be set to [ERANGE]. If the integer expression
 12321 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
 12322 floating-point exception shall be raised.

12323 **EXAMPLES**

12324 None.

12325 **APPLICATION USAGE**

12326 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
12327 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

12328 **RATIONALE**

12329 None.

12330 **FUTURE DIRECTIONS**

12331 None.

12332 **SEE ALSO**

12333 *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-2001,
12334 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

12335 **CHANGE HISTORY**

12336 First released in Issue 1. Derived from Issue 1 of the SVID.

12337 **Issue 5**

12338 The DESCRIPTION is updated to indicate how an application should check for an error. This
12339 text was previously published in the APPLICATION USAGE section.

12340 **Issue 6**

12341 The behavior for when the y argument is zero is now defined.

12342 The *fmodf()* and *fmodl()* functions are added for alignment with the ISO/IEC 9899:1999
12343 standard.

12344 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
12345 revised to align with the ISO/IEC 9899:1999 standard.

12346 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
12347 marked.

12348 **NAME**

12349 fmtmsg — display a message in the specified format on standard error and/or a system console

12350 **SYNOPSIS**

12351 XSI #include <fmtmsg.h>

```
12352       int fmtmsg(long classification, const char *label, int severity,
12353                const char *text, const char *action, const char *tag);
```

12354

12355 **DESCRIPTION**12356 The *fmtmsg()* function shall display messages in a specified format instead of the traditional
12357 *printf()* function.12358 Based on a message's classification component, *fmtmsg()* shall write a formatted message either
12359 to standard error, to the console, or to both.12360 A formatted message consists of up to five components as defined below. The component
12361 *classification* is not part of a message displayed to the user, but defines the source of the message
12362 and directs the display of the formatted message.

12363 *classification* Contains the sum of identifying values constructed from the constants defined
12364 below. Any one identifier from a subclass may be used in combination with a
12365 single identifier from a different subclass. Two or more identifiers from the
12366 same subclass should not be used together, with the exception of identifiers
12367 from the display subclass. (Both display subclass identifiers may be used so
12368 that messages can be displayed to both standard error and the system
12369 console.)

12370 **Major Classifications**12371 Identifies the source of the condition. Identifiers are: MM_HARD
12372 (hardware), MM_SOFT (software), and MM_FIRM (firmware).12373 **Message Source Subclassifications**12374 Identifies the type of software in which the problem is detected.
12375 Identifiers are: MM_APPL (application), MM_UTIL (utility), and
12376 MM_OPSYS (operating system).12377 **Display Subclassifications**12378 Indicates where the message is to be displayed. Identifiers are:
12379 MM_PRINT to display the message on the standard error stream,
12380 MM_CONSOLE to display the message on the system console. One or
12381 both identifiers may be used.12382 **Status Subclassifications**12383 Indicates whether the application can recover from the condition.
12384 Identifiers are: MM_RECOVER (recoverable) and MM_NRECOV (non-
12385 recoverable).12386 An additional identifier, MM_NULLMC, indicates that no classification
12387 component is supplied for the message.12388 *label* Identifies the source of the message. The format is two fields separated by a
12389 colon. The first field is up to 10 bytes, the second is up to 14 bytes.12390 *severity* Indicates the seriousness of the condition. Identifiers for the levels of *severity*
12391 are:

12392		MM_HALT	Indicates that the application has encountered a severe fault and is halting. Produces the string "HALT".
12393			
12394		MM_ERROR	Indicates that the application has detected a fault. Produces the string "ERROR".
12395			
12396		MM_WARNING	Indicates a condition that is out of the ordinary, that might be a problem, and should be watched. Produces the string "WARNING".
12397			
12398			
12399		MM_INFO	Provides information about a condition that is not in error. Produces the string "INFO".
12400			
12401		MM_NOSEV	Indicates that no severity level is supplied for the message.
12402	<i>text</i>		Describes the error condition that produced the message. The character string is not limited to a specific size. If the character string is empty, then the text produced is unspecified.
12403			
12404			
12405	<i>action</i>		Describes the first step to be taken in the error-recovery process. The <i>fmtmsg()</i> function precedes the action string with the prefix: "TO FIX:". The <i>action</i> string is not limited to a specific size.
12406			
12407			
12408	<i>tag</i>		An identifier that references on-line documentation for the message. Suggested usage is that <i>tag</i> includes the <i>label</i> and a unique identifying number. A sample <i>tag</i> is "XSI:cat:146".
12409			
12410			

12411 The *MSGVERB* environment variable (for message verbosity) shall determine for *fmtmsg()*
 12412 which message components it is to select when writing messages to standard error. The value of
 12413 *MSGVERB* shall be a colon-separated list of optional keywords. Valid keywords are: *label*,
 12414 *severity*, *text*, *action*, and *tag*. If *MSGVERB* contains a keyword for a component and the
 12415 component's value is not the component's null value, *fmtmsg()* shall include that component in
 12416 the message when writing the message to standard error. If *MSGVERB* does not include a
 12417 keyword for a message component, that component shall not be included in the display of the
 12418 message. The keywords may appear in any order. If *MSGVERB* is not defined, if its value is the
 12419 null string, if its value is not of the correct format, or if it contains keywords other than the valid
 12420 ones listed above, *fmtmsg()* shall select all components.

12421 *MSGVERB* shall determine which components are selected for display to standard error. All
 12422 message components shall be included in console messages.

12423 RETURN VALUE

12424 The *fmtmsg()* function shall return one of the following values:

12425	MM_OK	The function succeeded.
12426	MM_NOTOK	The function failed completely.
12427	MM_NOMSG	The function was unable to generate a message on standard error, but otherwise succeeded.
12428		
12429	MM_NOCON	The function was unable to generate a console message, but otherwise succeeded.
12430		

12431 ERRORS

12432 None.

12433 **EXAMPLES**12434 1. The following example of *fmtmsg()*:

```
12435     fmtmsg(MM_PRINT, "XSI:cat", MM_ERROR, "illegal option",
12436           "refer to cat in user's reference manual", "XSI:cat:001")
```

12437 produces a complete message in the specified message format:

```
12438     XSI:cat: ERROR: illegal option
12439     TO FIX: refer to cat in user's reference manual XSI:cat:001
```

12440 2. When the environment variable *MSGVERB* is set as follows:12441 *MSGVERB*=severity:text:action12442 and Example 1 is used, *fmtmsg()* produces:

```
12443     ERROR: illegal option
12444     TO FIX: refer to cat in user's reference manual
```

12445 **APPLICATION USAGE**

12446 One or more message components may be systematically omitted from messages generated by
 12447 an application by using the null value of the argument for that component.

12448 **RATIONALE**

12449 None.

12450 **FUTURE DIRECTIONS**

12451 None.

12452 **SEE ALSO**12453 *printf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <*fmtmsg.h*>12454 **CHANGE HISTORY**

12455 First released in Issue 4, Version 2.

12456 **Issue 5**

12457 Moved from X/OPEN UNIX extension to BASE.

12458 **NAME**

12459 fnmatch — match a filename or a pathname

12460 **SYNOPSIS**

12461 #include <fnmatch.h>

12462 int fnmatch(const char **pattern*, const char **string*, int *flags*);12463 **DESCRIPTION**

12464 The *fnmatch()* function shall match patterns as described in the Shell and Utilities volume of
 12465 IEEE Std 1003.1-2001, Section 2.13.1, Patterns Matching a Single Character, and Section 2.13.2,
 12466 Patterns Matching Multiple Characters. It checks the string specified by the *string* argument to
 12467 see if it matches the pattern specified by the *pattern* argument.

12468 The *flags* argument shall modify the interpretation of *pattern* and *string*. It is the bitwise-inclusive
 12469 OR of zero or more of the flags defined in <fnmatch.h>. If the FNM_PATHNAME flag is set in
 12470 *flags*, then a slash character ('/') in *string* shall be explicitly matched by a slash in *pattern*; it shall
 12471 not be matched by either the asterisk or question-mark special characters, nor by a bracket
 12472 expression. If the FNM_PATHNAME flag is not set, the slash character shall be treated as an
 12473 ordinary character.

12474 If FNM_NOESCAPE is not set in *flags*, a backslash character ('\ ') in *pattern* followed by any
 12475 other character shall match that second character in *string*. In particular, "\\\" shall match a
 12476 backslash in *string*. If FNM_NOESCAPE is set, a backslash character shall be treated as an
 12477 ordinary character.

12478 If FNM_PERIOD is set in *flags*, then a leading period (' . ') in *string* shall match a period in
 12479 *pattern*; as described by rule 2 in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section
 12480 2.13.3, Patterns Used for Filename Expansion where the location of “leading” is indicated by the
 12481 value of FNM_PATHNAME:

- 12482 • If FNM_PATHNAME is set, a period is “leading” if it is the first character in *string* or if it
 12483 immediately follows a slash.
- 12484 • If FNM_PATHNAME is not set, a period is “leading” only if it is the first character of *string*.

12485 If FNM_PERIOD is not set, then no special restrictions are placed on matching a period.

12486 **RETURN VALUE**

12487 If *string* matches the pattern specified by *pattern*, then *fnmatch()* shall return 0. If there is no
 12488 match, *fnmatch()* shall return FNM_NOMATCH, which is defined in <fnmatch.h>. If an error
 12489 occurs, *fnmatch()* shall return another non-zero value.

12490 **ERRORS**

12491 No errors are defined.

12492 **EXAMPLES**

12493 None.

12494 **APPLICATION USAGE**

12495 The *fnmatch()* function has two major uses. It could be used by an application or utility that
 12496 needs to read a directory and apply a pattern against each entry. The *find* utility is an example of
 12497 this. It can also be used by the *pax* utility to process its *pattern* operands, or by applications that
 12498 need to match strings in a similar manner.

12499 The name *fnmatch()* is intended to imply *filename* match, rather than *pathname* match. The default
 12500 action of this function is to match filenames, rather than pathnames, since it gives no special
 12501 significance to the slash character. With the FNM_PATHNAME flag, *fnmatch()* does match
 12502 pathnames, but without tilde expansion, parameter expansion, or special treatment for a period

12503 at the beginning of a filename.

12504 **RATIONALE**

12505 This function replaced the REG_FILENAME flag of *regcomp()* in early proposals of this volume
12506 of IEEE Std 1003.1-2001. It provides virtually the same functionality as the *regcomp()* and
12507 *regexexec()* functions using the REG_FILENAME and REG_FSLASH flags (the REG_FSLASH flag
12508 was proposed for *regcomp()*, and would have had the opposite effect from FNM_PATHNAME),
12509 but with a simpler function and less system overhead.

12510 **FUTURE DIRECTIONS**

12511 None.

12512 **SEE ALSO**

12513 *glob()*, *wordexp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <fnmatch.h>, the Shell
12514 and Utilities volume of IEEE Std 1003.1-2001

12515 **CHANGE HISTORY**

12516 First released in Issue 4. Derived from the ISO POSIX-2 standard.

12517 **Issue 5**

12518 Moved from POSIX2 C-language Binding to BASE.

12519 NAME

12520 `fopen` — open a stream

12521 SYNOPSIS

12522 `#include <stdio.h>`12523 `FILE *fopen(const char *restrict filename, const char *restrict mode);`

12524 DESCRIPTION

12525 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 12526 conflict between the requirements described here and the ISO C standard is unintentional. This
 12527 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

12528 The `fopen()` function shall open the file whose pathname is the string pointed to by `filename`, and
 12529 associates a stream with it.

12530 The `mode` argument points to a string. If the string is one of the following, the file shall be opened
 12531 in the indicated mode. Otherwise, the behavior is undefined.

12532	<code>r</code> or <code>rb</code>	Open file for reading.
12533	<code>w</code> or <code>wb</code>	Truncate to zero length or create file for writing.
12534	<code>a</code> or <code>ab</code>	Append; open or create file for writing at end-of-file.
12535	<code>r+</code> or <code>rb+</code> or <code>r+b</code>	Open file for update (reading and writing).
12536	<code>w+</code> or <code>wb+</code> or <code>w+b</code>	Truncate to zero length or create file for update.
12537	<code>a+</code> or <code>ab+</code> or <code>a+b</code>	Append; open or create file for update, writing at end-of-file.

12538 CX The character '`b`' shall have no effect, but is allowed for ISO C standard conformance. Opening
 12539 a file with read mode (`r` as the first character in the `mode` argument) shall fail if the file does not
 12540 exist or cannot be read.

12541 Opening a file with append mode (`a` as the first character in the `mode` argument) shall cause all
 12542 subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening
 12543 calls to `fseek()`.

12544 When a file is opened with update mode ('`+`' as the second or third character in the `mode`
 12545 argument), both input and output may be performed on the associated stream. However, the
 12546 application shall ensure that output is not directly followed by input without an intervening call
 12547 to `fflush()` or to a file positioning function (`fseek()`, `fsetpos()`, or `rewind()`), and input is not directly
 12548 followed by output without an intervening call to a file positioning function, unless the input
 12549 operation encounters end-of-file.

12550 When opened, a stream is fully buffered if and only if it can be determined not to refer to an
 12551 interactive device. The error and end-of-file indicators for the stream shall be cleared.

12552 CX If `mode` is `w`, `wb`, `a`, `ab`, `w+`, `wb+`, `w+b`, `a+`, `ab+`, or `a+b`, and the file did not previously exist, upon
 12553 successful completion, the `fopen()` function shall mark for update the `st_atime`, `st_ctime`, and
 12554 `st_mtime` fields of the file and the `st_ctime` and `st_mtime` fields of the parent directory.

12555 If `mode` is `w`, `wb`, `w+`, `wb+`, or `w+b`, and the file did previously exist, upon successful completion,
 12556 `fopen()` shall mark for update the `st_ctime` and `st_mtime` fields of the file. The `fopen()` function
 12557 shall allocate a file descriptor as `open()` does.

12558 XSI After a successful call to the `fopen()` function, the orientation of the stream shall be cleared, the
 12559 encoding rule shall be cleared, and the associated `mbstate_t` object shall be set to describe an
 12560 initial conversion state.

12561 CX The largest value that can be represented correctly in an object of type `off_t` shall be established
 12562 as the offset maximum in the open file description.

12563 RETURN VALUE

12564 Upon successful completion, *fopen()* shall return a pointer to the object controlling the stream.
 12565 CX Otherwise, a null pointer shall be returned, and *errno* shall be set to indicate the error.

12566 ERRORS

12567 The *fopen()* function shall fail if:

12568 CX [EACCES] Search permission is denied on a component of the path prefix, or the file
 12569 exists and the permissions specified by *mode* are denied, or the file does not
 12570 exist and write permission is denied for the parent directory of the file to be
 12571 created.

12572 CX [EINTR] A signal was caught during *fopen()*.

12573 CX [EISDIR] The named file is a directory and *mode* requires write access.

12574 CX [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 12575 argument.

12576 CX [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

12577 CX [ENAMETOOLONG]
 12578 The length of the *filename* argument exceeds {PATH_MAX} or a pathname
 12579 component is longer than {NAME_MAX}.

12580 CX [ENFILE] The maximum allowable number of files is currently open in the system.

12581 CX [ENOENT] A component of *filename* does not name an existing file or *filename* is an empty
 12582 string.

12583 CX [ENOSPC] The directory or file system that would contain the new file cannot be
 12584 expanded, the file does not exist, and the file was to be created.

12585 CX [ENOTDIR] A component of the path prefix is not a directory.

12586 CX [ENXIO] The named file is a character special or block special file, and the device
 12587 associated with this special file does not exist.

12588 CX [EOVERFLOW] The named file is a regular file and the size of the file cannot be represented
 12589 correctly in an object of type `off_t`.

12590 CX [EROFS] The named file resides on a read-only file system and *mode* requires write
 12591 access.

12592 The *fopen()* function may fail if:

12593 CX [EINVAL] The value of the *mode* argument is not valid.

12594 CX [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 12595 resolution of the *path* argument.

12596 CX [EMFILE] {FOPEN_MAX} streams are currently open in the calling process.

12597 CX [EMFILE] {STREAM_MAX} streams are currently open in the calling process.

12598 CX [ENAMETOOLONG]
 12599 Pathname resolution of a symbolic link produced an intermediate result
 12600 whose length exceeds {PATH_MAX}.

12601	CX	[ENOMEM]	Insufficient storage space is available.
12602	CX	[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed and <i>mode</i> requires write access.
12603			

12604 **EXAMPLES**12605 **Opening a File**

12606 The following example tries to open the file named **file** for reading. The *fopen()* function returns
 12607 a file pointer that is used in subsequent *fgets()* and *fclose()* calls. If the program cannot open the
 12608 file, it just ignores it.

```

12609 #include <stdio.h>
12610 ...
12611 FILE *fp;
12612 ...
12613 void rgrep(const char *file)
12614 {
12615     ...
12616     if ((fp = fopen(file, "r")) == NULL)
12617         return;
12618     ...
12619 }
```

12620 **APPLICATION USAGE**

12621 None.

12622 **RATIONALE**

12623 None.

12624 **FUTURE DIRECTIONS**

12625 None.

12626 **SEE ALSO**

12627 *fclose()*, *fdopen()*, *freopen()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<stdio.h>**

12628 **CHANGE HISTORY**

12629 First released in Issue 1. Derived from Issue 1 of the SVID.

12630 **Issue 5**

12631 Large File Summit extensions are added.

12632 **Issue 6**

12633 Extensions beyond the ISO C standard are marked.

12634 The following new requirements on POSIX implementations derive from alignment with the
 12635 Single UNIX Specification:

- 12636 • In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file
 12637 description. This change is to support large files.
- 12638 • In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support
 12639 large files.
- 12640 • The [ELOOP] mandatory error condition is added.
- 12641 • The [EINVAL], [EMFILE], [ENAMETOOLONG], [ENOMEM], and [ETXTBSY] optional error
 12642 conditions are added.

- 12643 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
- 12644 The following changes are made for alignment with the ISO/IEC 9899: 1999 standard:
- 12645 • The prototype for *fopen()* is updated.
 - 12646 • The DESCRIPTION is updated to note that if the argument *mode* points to a string other than
 - 12647 those listed, then the behavior is undefined.
- 12648 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
- 12649 [ELOOP] error condition is added.

12650 NAME

12651 fork — create a new process

12652 SYNOPSIS

12653 #include <unistd.h>

12654 pid_t fork(void);

12655 DESCRIPTION

12656 The *fork()* function shall create a new process. The new process (child process) shall be an exact
 12657 copy of the calling process (parent process) except as detailed below:

- 12658 • The child process shall have a unique process ID.
- 12659 • The child process ID also shall not match any active process group ID.
- 12660 • The child process shall have a different parent process ID, which shall be the process ID of
 12661 the calling process.
- 12662 • The child process shall have its own copy of the parent's file descriptors. Each of the child's
 12663 file descriptors shall refer to the same open file description with the corresponding file
 12664 descriptor of the parent.
- 12665 • The child process shall have its own copy of the parent's open directory streams. Each open
 12666 directory stream in the child process may share directory stream positioning with the
 12667 corresponding directory stream of the parent.
- 12668 XSI • The child process shall have its own copy of the parent's message catalog descriptors.
- 12669 • The child process' values of *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* shall be set to 0.
- 12670 • The time left until an alarm clock signal shall be reset to zero, and the alarm, if any, shall be
 12671 canceled; see *alarm()*.
- 12672 XSI • All *semadj* values shall be cleared.
- 12673 • File locks set by the parent process shall not be inherited by the child process.
- 12674 • The set of signals pending for the child process shall be initialized to the empty set.
- 12675 XSI • Interval timers shall be reset in the child process.
- 12676 SEM • Any semaphores that are open in the parent process shall also be open in the child process.
- 12677 ML • The child process shall not inherit any address space memory locks established by the parent
 12678 process via calls to *mlockall()* or *mlock()*.
- 12679 MF|SHM • Memory mappings created in the parent shall be retained in the child process.
 12680 MAP_PRIVATE mappings inherited from the parent shall also be MAP_PRIVATE mappings
 12681 in the child, and any modifications to the data in these mappings made by the parent prior to
 12682 calling *fork()* shall be visible to the child. Any modifications to the data in MAP_PRIVATE
 12683 mappings made by the parent after *fork()* returns shall be visible only to the parent.
 12684 Modifications to the data in MAP_PRIVATE mappings made by the child shall be visible only
 12685 to the child.
- 12686 PS • For the SCHED_FIFO and SCHED_RR scheduling policies, the child process shall inherit the
 12687 policy and priority settings of the parent process during a *fork()* function. For other
 12688 scheduling policies, the policy and priority settings on *fork()* are implementation-defined.
- 12689 TMR • Per-process timers created by the parent shall not be inherited by the child process.
- 12690 MSG • The child process shall have its own copy of the message queue descriptors of the parent.
 12691 Each of the message descriptors of the child shall refer to the same open message queue

12692		description as the corresponding message descriptor of the parent.
12693 AIO		• No asynchronous input or asynchronous output operations shall be inherited by the child
12694		process.
12695		• A process shall be created with a single thread. If a multi-threaded process calls <i>fork()</i> , the
12696		new process shall contain a replica of the calling thread and its entire address space, possibly
12697		including the states of mutexes and other resources. Consequently, to avoid errors, the child
12698		process may only execute async-signal-safe operations until such time as one of the <i>exec</i>
12699 THR		functions is called. Fork handlers may be established by means of the <i>pthread_atfork()</i>
12700		function in order to maintain application invariants across <i>fork()</i> calls.
12701 TRC TRI		• If the Trace option and the Trace Inherit option are both supported:
12702		If the calling process was being traced in a trace stream that had its inheritance policy set to
12703		POSIX_TRACE_INHERITED, the child process shall be traced into that trace stream, and the
12704		child process shall inherit the parent's mapping of trace event names to trace event type
12705		identifiers. If the trace stream in which the calling process was being traced had its
12706		inheritance policy set to POSIX_TRACE_CLOSE_FOR_CHILD, the child process shall not be
12707		traced into that trace stream. The inheritance policy is set by a call to the
12708		<i>posix_trace_attr_setinherited()</i> function.
12709 TRC		• If the Trace option is supported, but the Trace Inherit option is not supported:
12710		The child process shall not be traced into any of the trace streams of its parent process.
12711 TRC		• If the Trace option is supported, the child process of a trace controller process shall not
12712		control the trace streams controlled by its parent process.
12713 CPT		• The initial value of the CPU-time clock of the child process shall be set to zero.
12714 TCT		• The initial value of the CPU-time clock of the single thread of the child process shall be set to
12715		zero.
12716		All other process characteristics defined by IEEE Std 1003.1-2001 shall be the same in the parent
12717		and child processes. The inheritance of process characteristics not defined by
12718		IEEE Std 1003.1-2001 is unspecified by IEEE Std 1003.1-2001.
12719		After <i>fork()</i> , both the parent and the child processes shall be capable of executing independently
12720		before either one terminates.
12721	RETURN VALUE	
12722		Upon successful completion, <i>fork()</i> shall return 0 to the child process and shall return the
12723		process ID of the child process to the parent process. Both processes shall continue to execute
12724		from the <i>fork()</i> function. Otherwise, -1 shall be returned to the parent process, no child process
12725		shall be created, and <i>errno</i> shall be set to indicate the error.
12726	ERRORS	
12727		The <i>fork()</i> function shall fail if:
12728	[EAGAIN]	The system lacked the necessary resources to create another process, or the
12729		system-imposed limit on the total number of processes under execution
12730		system-wide or by a single user {CHILD_MAX} would be exceeded.
12731		The <i>fork()</i> function may fail if:
12732	[ENOMEM]	Insufficient storage space is available.

12733 **EXAMPLES**

12734 None.

12735 **APPLICATION USAGE**

12736 None.

12737 **RATIONALE**

12738 Many historical implementations have timing windows where a signal sent to a process group
 12739 (for example, an interactive SIGINT) just prior to or during execution of *fork()* is delivered to the
 12740 parent following the *fork()* but not to the child because the *fork()* code clears the child's set of
 12741 pending signals. This volume of IEEE Std 1003.1-2001 does not require, or even permit, this
 12742 behavior. However, it is pragmatic to expect that problems of this nature may continue to exist
 12743 in implementations that appear to conform to this volume of IEEE Std 1003.1-2001 and pass
 12744 available verification suites. This behavior is only a consequence of the implementation failing to
 12745 make the interval between signal generation and delivery totally invisible. From the
 12746 application's perspective, a *fork()* call should appear atomic. A signal that is generated prior to
 12747 the *fork()* should be delivered prior to the *fork()*. A signal sent to the process group after the
 12748 *fork()* should be delivered to both parent and child. The implementation may actually initialize
 12749 internal data structures corresponding to the child's set of pending signals to include signals
 12750 sent to the process group during the *fork()*. Since the *fork()* call can be considered as atomic
 12751 from the application's perspective, the set would be initialized as empty and such signals would
 12752 have arrived after the *fork()*; see also <signal.h>.

12753 One approach that has been suggested to address the problem of signal inheritance across *fork()*
 12754 is to add an [EINTR] error, which would be returned when a signal is detected during the call.
 12755 While this is preferable to losing signals, it was not considered an optimal solution. Although it
 12756 is not recommended for this purpose, such an error would be an allowable extension for an
 12757 implementation.

12758 The [ENOMEM] error value is reserved for those implementations that detect and distinguish
 12759 such a condition. This condition occurs when an implementation detects that there is not enough
 12760 memory to create the process. This is intended to be returned when [EAGAIN] is inappropriate
 12761 because there can never be enough memory (either primary or secondary storage) to perform the
 12762 operation. Since *fork()* duplicates an existing process, this must be a condition where there is
 12763 sufficient memory for one such process, but not for two. Many historical implementations
 12764 actually return [ENOMEM] due to temporary lack of memory, a case that is not generally
 12765 distinct from [EAGAIN] from the perspective of a conforming application.

12766 Part of the reason for including the optional error [ENOMEM] is because the SVID specifies it
 12767 and it should be reserved for the error condition specified there. The condition is not applicable
 12768 on many implementations.

12769 IEEE Std 1003.1-1988 neglected to require concurrent execution of the parent and child of *fork()*.
 12770 A system that single-threads processes was clearly not intended and is considered an
 12771 unacceptable "toy implementation" of this volume of IEEE Std 1003.1-2001. The only objection
 12772 anticipated to the phrase "executing independently" is testability, but this assertion should be
 12773 testable. Such tests require that both the parent and child can block on a detectable action of the
 12774 other, such as a write to a pipe or a signal. An interactive exchange of such actions should be
 12775 possible for the system to conform to the intent of this volume of IEEE Std 1003.1-2001.

12776 The [EAGAIN] error exists to warn applications that such a condition might occur. Whether it
 12777 occurs or not is not in any practical sense under the control of the application because the
 12778 condition is usually a consequence of the user's use of the system, not of the application's code.
 12779 Thus, no application can or should rely upon its occurrence under any circumstances, nor
 12780 should the exact semantics of what concept of "user" is used be of concern to the application
 12781 writer. Validation writers should be cognizant of this limitation.

There are two reasons why POSIX programmers call *fork()*. One reason is to create a new thread of control within the same program (which was originally only possible in POSIX by creating a new process); the other is to create a new process running a different program. In the latter case, the call to *fork()* is soon followed by a call to one of the *exec* functions.

The general problem with making *fork()* work in a multi-threaded world is what to do with all of the threads. There are two alternatives. One is to copy all of the threads into the new process. This causes the programmer or implementation to deal with threads that are suspended on system calls or that might be about to execute system calls that should not be executed in the new process. The other alternative is to copy only the thread that calls *fork()*. This creates the difficulty that the state of process-local resources is usually held in process memory. If a thread that is not calling *fork()* holds a resource, that resource is never released in the child process because the thread whose job it is to release the resource does not exist in the child process.

When a programmer is writing a multi-threaded program, the first described use of *fork()*, creating new threads in the same program, is provided by the *pthread_create()* function. The *fork()* function is thus used only to run new programs, and the effects of calling functions that require certain resources between the call to *fork()* and the call to an *exec* function are undefined.

The addition of the *forkall()* function to the standard was considered and rejected. The *forkall()* function lets all the threads in the parent be duplicated in the child. This essentially duplicates the state of the parent in the child. This allows threads in the child to continue processing and allows locks and the state to be preserved without explicit *pthread_atfork()* code. The calling process has to ensure that the threads processing state that is shared between the parent and child (that is, file descriptors or MAP_SHARED memory) behaves properly after *forkall()*. For example, if a thread is reading a file descriptor in the parent when *forkall()* is called, then two threads (one in the parent and one in the child) are reading the file descriptor after the *forkall()*. If this is not desired behavior, the parent process has to synchronize with such threads before calling *forkall()*.

When *forkall()* is called, threads, other than the calling thread, that are in functions that can return with an [EINTR] error may have those functions return [EINTR] if the implementation cannot ensure that the function behaves correctly in the parent and child. In particular, *pthread_cond_wait()* and *pthread_cond_timedwait()* need to return in order to ensure that the condition has not changed. These functions can be awakened by a spurious condition wakeup rather than returning [EINTR].

12814 FUTURE DIRECTIONS

12815 None.

12816 SEE ALSO

12817 *alarm()*, *exec*, *fcntl()*, *posix_trace_attr_getinherited()*, *posix_trace_trid_eventid_open()*, *semop()*,
12818 *signal()*, *times()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

12819 CHANGE HISTORY

12820 First released in Issue 1. Derived from Issue 1 of the SVID.

12821 Issue 5

12822 The DESCRIPTION is changed for alignment with the POSIX Realtime Extension and the POSIX
12823 Threads Extension.

12824 Issue 6

12825 The following new requirements on POSIX implementations derive from alignment with the
12826 Single UNIX Specification:

- 12827 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
12828 required for conforming implementations of previous POSIX specifications, it was not

12829 required for UNIX applications.

12830 The following changes were made to align with the IEEE P1003.1a draft standard:

- 12831 • The effect of *fork()* on a pending alarm call in the child process is clarified.

12832 The description of CPU-time clock semantics is added for alignment with IEEE Std 1003.1d-1999.

12833 The description of tracing semantics is added for alignment with IEEE Std 1003.1q-2000.

12834 **NAME**

12835 fpathconf, pathconf — get configurable pathname variables

12836 **SYNOPSIS**

12837 #include <unistd.h>

12838 long fpathconf(int *fildes*, int *name*);12839 long pathconf(const char **path*, int *name*);12840 **DESCRIPTION**12841 The *fpathconf()* and *pathconf()* functions shall determine the current value of a configurable limit
12842 or option (*variable*) that is associated with a file or directory.12843 For *pathconf()*, the *path* argument points to the pathname of a file or directory.12844 For *fpathconf()*, the *fildes* argument is an open file descriptor.

12845 The *name* argument represents the variable to be queried relative to that file or directory.
 12846 Implementations shall support all of the variables listed in the following table and may support
 12847 others. The variables in the following table come from <limits.h> or <unistd.h> and the
 12848 symbolic constants, defined in <unistd.h>, are the corresponding values used for *name*. Support
 12849 for some pathname configuration variables is dependent on implementation options (see
 12850 shading and margin codes in the table below). Where an implementation option is not
 12851 supported, the variable need not be supported.

12852

12853

12854

12855

12856

12857

12858

12859

12860

12861 ADV

12862 ADV

12863 ADV

12864 ADV

12865 ADV

12866

12867

12868

12869

12870

12871

12872

Variable	Value of <i>name</i>	Requirements
{FILESIZEBITS}	_PC_FILESIZEBITS	3, 4
{LINK_MAX}	_PC_LINK_MAX	1
{MAX_CANON}	_PC_MAX_CANON	2
{MAX_INPUT}	_PC_MAX_INPUT	2
{NAME_MAX}	_PC_NAME_MAX	3, 4
{PATH_MAX}	_PC_PATH_MAX	4, 5
{PIPE_BUF}	_PC_PIPE_BUF	6
{POSIX_ALLOC_SIZE_MIN}	_PC_ALLOC_SIZE_MIN	
{POSIX_REC_INCR_XFER_SIZE}	_PC_REC_INCR_XFER_SIZE	
{POSIX_REC_MAX_XFER_SIZE}	_PC_REC_MAX_XFER_SIZE	
{POSIX_REC_MIN_XFER_SIZE}	_PC_REC_MIN_XFER_SIZE	
{POSIX_REC_XFER_ALIGN}	_PC_REC_XFER_ALIGN	
{SYMLINK_MAX}	_PC_SYMLINK_MAX	4, 9
_POSIX_CHOWN_RESTRICTED	_PC_CHOWN_RESTRICTED	7
_POSIX_NO_TRUNC	_PC_NO_TRUNC	3, 4
_POSIX_VDISABLE	_PC_VDISABLE	2
_POSIX_ASYNC_IO	_PC_ASYNC_IO	8
_POSIX_PRIO_IO	_PC_PRIO_IO	8
_POSIX_SYNC_IO	_PC_SYNC_IO	8

12873 **Requirements**

- 12874 1. If *path* or *filde*s refers to a directory, the value returned shall apply to the directory itself.
- 12875 2. If *path* or *filde*s does not refer to a terminal file, it is unspecified whether an implementation
- 12876 supports an association of the variable name with the specified file.
- 12877 3. If *path* or *filde*s refers to a directory, the value returned shall apply to filenames within the
- 12878 directory.
- 12879 4. If *path* or *filde*s does not refer to a directory, it is unspecified whether an implementation
- 12880 supports an association of the variable name with the specified file.
- 12881 5. If *path* or *filde*s refers to a directory, the value returned shall be the maximum length of a
- 12882 relative pathname when the specified directory is the working directory.
- 12883 6. If *path* refers to a FIFO, or *filde*s refers to a pipe or FIFO, the value returned shall apply to
- 12884 the referenced object. If *path* or *filde*s refers to a directory, the value returned shall apply to
- 12885 any FIFO that exists or can be created within the directory. If *path* or *filde*s refers to any
- 12886 other type of file, it is unspecified whether an implementation supports an association of
- 12887 the variable name with the specified file.
- 12888 7. If *path* or *filde*s refers to a directory, the value returned shall apply to any files, other than
- 12889 directories, that exist or can be created within the directory.
- 12890 8. If *path* or *filde*s refers to a directory, it is unspecified whether an implementation supports
- 12891 an association of the variable name with the specified file.
- 12892 9. If *path* or *filde*s refers to a directory, the value returned shall be the maximum length of the
- 12893 string that a symbolic link in that directory can contain.

12894 **RETURN VALUE**

12895 If *name* is an invalid value, both *pathconf()* and *fpathconf()* shall return `-1` and set *errno* to

12896 indicate the error.

12897 If the variable corresponding to *name* has no limit for the *path* or file descriptor, both *pathconf()*

12898 and *fpathconf()* shall return `-1` without changing *errno*. If the implementation needs to use *path*

12899 to determine the value of *name* and the implementation does not support the association of *name*

12900 with the file specified by *path*, or if the process did not have appropriate privileges to query the

12901 file specified by *path*, or *path* does not exist, *pathconf()* shall return `-1` and set *errno* to indicate the

12902 error.

12903 If the implementation needs to use *filde*s to determine the value of *name* and the implementation

12904 does not support the association of *name* with the file specified by *filde*s, or if *filde*s is an invalid

12905 file descriptor, *fpathconf()* shall return `-1` and set *errno* to indicate the error.

12906 Otherwise, *pathconf()* or *fpathconf()* shall return the current variable value for the file or

12907 directory without changing *errno*. The value returned shall not be more restrictive than the

12908 corresponding value available to the application when it was compiled with the

12909 implementation's `<limits.h>` or `<unistd.h>`.

12910 **ERRORS**

12911 The *pathconf()* function shall fail if:

- | | | |
|-------|----------|--|
| 12912 | [EINVAL] | The value of <i>name</i> is not valid. |
| 12913 | [ELOOP] | A loop exists in symbolic links encountered during resolution of the <i>path</i> |
| 12914 | | argument. |

12915 The *pathconf()* function may fail if:

12916	[EACCES]	Search permission is denied for a component of the path prefix.
12917	[EINVAL]	The implementation does not support an association of the variable <i>name</i> with the specified file.
12918		
12919	[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the <i>path</i> argument.
12920		
12921	[ENAMETOOLONG]	The length of the <i>path</i> argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
12922		
12923		
12924	[ENAMETOOLONG]	As a result of encountering a symbolic link in resolution of the <i>path</i> argument, the length of the substituted pathname string exceeded {PATH_MAX}.
12925		
12926		
12927	[ENOENT]	A component of <i>path</i> does not name an existing file or <i>path</i> is an empty string.
12928	[ENOTDIR]	A component of the path prefix is not a directory.
12929	The <i>fpathconf()</i> function shall fail if:	
12930	[EINVAL]	The value of <i>name</i> is not valid.
12931	The <i>fpathconf()</i> function may fail if:	
12932	[EBADF]	The <i>fdes</i> argument is not a valid file descriptor.
12933	[EINVAL]	The implementation does not support an association of the variable <i>name</i> with the specified file.
12934		

12935 EXAMPLES

12936 None.

12937 APPLICATION USAGE

12938 None.

12939 RATIONALE

12940 The *pathconf()* function was proposed immediately after the *sysconf()* function when it was realized that some configurable values may differ across file system, directory, or device boundaries.

12943 For example, {NAME_MAX} frequently changes between System V and BSD-based file systems; 12944 System V uses a maximum of 14, BSD 255. On an implementation that provides both types of file 12945 systems, an application would be forced to limit all pathname components to 14 bytes, as this 12946 would be the value specified in <limits.h> on such a system.

12947 Therefore, various useful values can be queried on any pathname or file descriptor, assuming 12948 that the appropriate permissions are in place.

12949 The value returned for the variable {PATH_MAX} indicates the longest relative pathname that 12950 could be given if the specified directory is the process' current working directory. A process may 12951 not always be able to generate a name that long and use it if a subdirectory in the pathname 12952 crosses into a more restrictive file system.

12953 The value returned for the variable _POSIX_CHOWN_RESTRICTED also applies to directories 12954 that do not have file systems mounted on them. The value may change when crossing a mount 12955 point, so applications that need to know should check for each directory. (An even easier check 12956 is to try the *chown()* function and look for an error in case it happens.)

12957 Unlike the values returned by *sysconf()*, the pathname-oriented variables are potentially more 12958 volatile and are not guaranteed to remain constant throughout the process' lifetime. For

example, in between two calls to *pathconf()*, the file system in question may have been unmounted and remounted with different characteristics.

Also note that most of the errors are optional. If one of the variables always has the same value on an implementation, the implementation need not look at *path* or *filde*s to return that value and is, therefore, not required to detect any of the errors except the meaning of [EINVAL] that indicates that the value of *name* is not valid for that variable.

If the value of any of the limits is unspecified (logically infinite), they will not be defined in **<limits.h>** and the *pathconf()* and *fpathconf()* functions return -1 without changing *errno*. This can be distinguished from the case of giving an unrecognized *name* argument because *errno* is set to [EINVAL] in this case.

Since -1 is a valid return value for the *pathconf()* and *fpathconf()* functions, applications should set *errno* to zero before calling them and check *errno* only if the return value is -1.

For the case of {SYMLINK_MAX}, since both *pathconf()* and *open()* follow symbolic links, there is no way that *path* or *filde*s could refer to a symbolic link.

12973 FUTURE DIRECTIONS

12974 None.

12975 SEE ALSO

12976 *confstr()*, *sysconf()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<limits.h>**, **<unistd.h>**,
12977 the Shell and Utilities volume of IEEE Std 1003.1-2001

12978 CHANGE HISTORY

12979 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

12980 Issue 5

12981 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

12982 Large File Summit extensions are added.

12983 Issue 6

12984 The following new requirements on POSIX implementations derive from alignment with the
12985 Single UNIX Specification:

- 12986 • The DESCRIPTION is updated to include {FILESIZEBITS}.
- 12987 • The [ELOOP] mandatory error condition is added.
- 12988 • A second [ENAMETOOLONG] is added as an optional error condition.

12989 The following changes were made to align with the IEEE P1003.1a draft standard:

- 12990 • The _PC_SYMLINK_MAX entry is added to the table in the DESCRIPTION.

12991 The following *pathconf()* variables and their associated names are added for alignment with
12992 IEEE Std 1003.1d-1999:

12993 {POSIX_ALLOC_SIZE_MIN}
12994 {POSIX_REC_INCR_XFER_SIZE}
12995 {POSIX_REC_MAX_XFER_SIZE}
12996 {POSIX_REC_MIN_XFER_SIZE}
12997 {POSIX_REC_XFER_ALIGN}

12998 **NAME**

12999 fpclassify — classify real floating type

13000 **SYNOPSIS**

13001 #include <math.h>

13002 int fpclassify(real-floating x);

13003 **DESCRIPTION**

13004 cx The functionality described on this reference page is aligned with the ISO C standard. Any
13005 conflict between the requirements described here and the ISO C standard is unintentional. This
13006 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

13007 The *fpclassify()* macro shall classify its argument value as NaN, infinite, normal, subnormal,
13008 zero, or into another implementation-defined category. First, an argument represented in a
13009 format wider than its semantic type is converted to its semantic type. Then classification is based
13010 on the type of the argument.

13011 **RETURN VALUE**

13012 The *fpclassify()* macro shall return the value of the number classification macro appropriate to
13013 the value of its argument.

13014 **ERRORS**

13015 No errors are defined.

13016 **EXAMPLES**

13017 None.

13018 **APPLICATION USAGE**

13019 None.

13020 **RATIONALE**

13021 None.

13022 **FUTURE DIRECTIONS**

13023 None.

13024 **SEE ALSO**

13025 *isfinite()*, *isinf()*, *isnan()*, *isnormal()*, *signbit()*, the Base Definitions volume of
13026 IEEE Std 1003.1-2001, <math.h>

13027 **CHANGE HISTORY**

13028 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

13029 NAME

13030 fprintf, printf, snprintf, sprintf — print formatted output

13031 SYNOPSIS

13032 #include <stdio.h>

13033 int fprintf(FILE *restrict stream, const char *restrict format, ...);

13034 int printf(const char *restrict format, ...);

13035 int snprintf(char *restrict s, size_t n,

13036 const char *restrict format, ...);

13037 int sprintf(char *restrict s, const char *restrict format, ...);

13038 DESCRIPTION

13039 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 13040 conflict between the requirements described here and the ISO C standard is unintentional. This
 13041 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

13042 The *fprintf()* function shall place output on the named output *stream*. The *printf()* function shall
 13043 place output on the standard output stream *stdout*. The *sprintf()* function shall place output
 13044 followed by the null byte, `'\0'`, in consecutive bytes starting at *s*; it is the user's responsibility
 13045 to ensure that enough space is available.

13046 The *snprintf()* function shall be equivalent to *sprintf()*, with the addition of the *n* argument
 13047 which states the size of the buffer referred to by *s*. If *n* is zero, nothing shall be written and *s* may
 13048 be a null pointer. Otherwise, output bytes beyond the *n*-1st shall be discarded instead of being
 13049 written to the array, and a null byte is written at the end of the bytes actually written into the
 13050 array.

13051 If copying takes place between objects that overlap as a result of a call to *sprintf()* or *snprintf()*,
 13052 the results are undefined.

13053 Each of these functions converts, formats, and prints its arguments under control of the *format*.
 13054 The *format* is a character string, beginning and ending in its initial shift state, if any. The *format* is
 13055 composed of zero or more directives: *ordinary characters*, which are simply copied to the output
 13056 stream, and *conversion specifications*, each of which shall result in the fetching of zero or more
 13057 arguments. The results are undefined if there are insufficient arguments for the *format*. If the
 13058 *format* is exhausted while arguments remain, the excess arguments shall be evaluated but are
 13059 otherwise ignored.

13060 XSI Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than
 13061 to the next unused argument. In this case, the conversion specifier character `%` (see below) is
 13062 replaced by the sequence `"%n$"`, where *n* is a decimal integer in the range `[1,{NL_ARGMAX}]`,
 13063 giving the position of the argument in the argument list. This feature provides for the definition
 13064 of format strings that select arguments in an order appropriate to specific languages (see the
 13065 EXAMPLES section).

13066 The *format* can contain either numbered argument conversion specifications (that is, `"%n$"` and
 13067 `"*m$"`), or unnumbered argument conversion specifications (that is, `%` and `*`), but not both. The
 13068 only exception to this is that `%%` can be mixed with the `"%n$"` form. The results of mixing
 13069 numbered and unnumbered argument specifications in a *format* string are undefined. When
 13070 numbered argument specifications are used, specifying the *N*th argument requires that all the
 13071 leading arguments, from the first to the (*N*-1)th, are specified in the format string.

13072 In format strings containing the `"%n$"` form of conversion specification, numbered arguments
 13073 in the argument list can be referenced from the format string as many times as required.

13074 In format strings containing the `%` form of conversion specification, each conversion specification
 13075 uses the first unused argument in the argument list.

13076 CX	All forms of the <i>fprintf()</i> functions allow for the insertion of a language-dependent radix character in the output string. The radix character is defined in the program's locale (category <i>LC_NUMERIC</i>). In the POSIX locale, or in a locale where the radix character is not defined, the radix character shall default to a period ('.').
13077	
13078	
13079	
13080 XSI	Each conversion specification is introduced by the '%' character or by the character sequence
13081	"%n\$", after which the following appear in sequence:
13082	<ul style="list-style-type: none"> • Zero or more <i>flags</i> (in any order), which modify the meaning of the conversion specification.
13083	<ul style="list-style-type: none"> • An optional minimum <i>field width</i>. If the converted value has fewer bytes than the field width, it shall be padded with spaces by default on the left; it shall be padded on the right if
13084	the left-adjustment flag ('-'), described below, is given to the field width. The field width
13085	takes the form of an asterisk ('*'), described below, or a decimal integer.
13086	
13087	<ul style="list-style-type: none"> • An optional <i>precision</i> that gives the minimum number of digits to appear for the d, i, o, u, x,
13088	and X conversion specifiers; the number of digits to appear after the radix character for the a,
13089	A, e, E, f, and F conversion specifiers; the maximum number of significant digits for the g
13090	and G conversion specifiers; or the maximum number of bytes to be printed from a string in
13091 XSI	the s and S conversion specifiers. The precision takes the form of a period ('.') followed
13092	either by an asterisk ('*'), described below, or an optional decimal digit string, where a null
13093	digit string is treated as zero. If a precision appears with any other conversion specifier, the
13094	behavior is undefined.
13095	<ul style="list-style-type: none"> • An optional length modifier that specifies the size of the argument.
13096	<ul style="list-style-type: none"> • A <i>conversion specifier</i> character that indicates the type of conversion to be applied.
13097	A field width, or precision, or both, may be indicated by an asterisk ('*'). In this case an
13098	argument of type <i>int</i> supplies the field width or precision. Applications shall ensure that
13099	arguments specifying field width, or precision, or both appear in that order before the argument,
13100	if any, to be converted. A negative field width is taken as a '-' flag followed by a positive field
13101 XSI	width. A negative precision is taken as if the precision were omitted. In format strings
13102	containing the "%n\$" form of a conversion specification, a field width or precision may be
13103	indicated by the sequence "%m\$", where <i>m</i> is a decimal integer in the range [1,{NL_ARGMAX}]
13104	giving the position in the argument list (after the <i>format</i> argument) of an integer argument
13105	containing the field width or precision, for example:
13106	<pre>printf("%1\$d:%2\$.*3\$d:%4\$.*3\$d\n", hour, min, precision, sec);</pre>
13107	The flag characters and their meanings are:
13108 XSI	' The integer portion of the result of a decimal conversion (%i, %d, %u, %f, %F, %g, or %G)
13109	shall be formatted with thousands' grouping characters. For other conversions the
13110	behavior is undefined. The non-monetary grouping character is used.
13111	- The result of the conversion shall be left-justified within the field. The conversion is
13112	right-justified if this flag is not specified.
13113	+ The result of a signed conversion shall always begin with a sign ('+' or '-'). The
13114	conversion shall begin with a sign only when a negative value is converted if this flag is
13115	not specified.
13116	<space> If the first character of a signed conversion is not a sign or if a signed conversion results
13117	in no characters, a <space> shall be prefixed to the result. This means that if the
13118	<space> and '+' flags both appear, the <space> flag shall be ignored.
13119	# Specifies that the value is to be converted to an alternative form. For o conversion, it
13120	increases the precision (if necessary) to force the first digit of the result to be zero. For x

13121 or X conversion specifiers, a non-zero result shall have 0x (or 0X) prefixed to it. For a, A,
 13122 e, E, f, F, g, and G conversion specifiers, the result shall always contain a radix
 13123 character, even if no digits follow the radix character. Without this flag, a radix
 13124 character appears in the result of these conversions only if a digit follows it. For g and G
 13125 conversion specifiers, trailing zeros shall *not* be removed from the result as they
 13126 normally are. For other conversion specifiers, the behavior is undefined.

13127 0 For d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversion specifiers, leading zeros
 13128 (following any indication of sign or base) are used to pad to the field width; no space
 13129 padding is performed. If the '0' and '-' flags both appear, the '0' flag is ignored. For
 13130 d, i, o, u, x, and X conversion specifiers, if a precision is specified, the '0' flag is
 13131 XSI ignored. If the '0' and ' ' flags both appear, the grouping characters are inserted
 13132 before zero padding. For other conversions, the behavior is undefined.

13133 The length modifiers and their meanings are:

13134 hh Specifies that a following d, i, o, u, x, or X conversion specifier applies to a **signed char**
 13135 or **unsigned char** argument (the argument will have been promoted according to the
 13136 integer promotions, but its value shall be converted to **signed char** or **unsigned char**
 13137 before printing); or that a following n conversion specifier applies to a pointer to a
 13138 **signed char** argument.

13139 h Specifies that a following d, i, o, u, x, or X conversion specifier applies to a **short** or
 13140 **unsigned short** argument (the argument will have been promoted according to the
 13141 integer promotions, but its value shall be converted to **short** or **unsigned short** before
 13142 printing); or that a following n conversion specifier applies to a pointer to a **short**
 13143 argument.

13144 l (ell) Specifies that a following d, i, o, u, x, or X conversion specifier applies to a **long** or
 13145 **unsigned long** argument; that a following n conversion specifier applies to a pointer to
 13146 a **long** argument; that a following c conversion specifier applies to a **wint_t** argument;
 13147 that a following s conversion specifier applies to a pointer to a **wchar_t** argument; or
 13148 has no effect on a following a, A, e, E, f, F, g, or G conversion specifier.

13149 ll (ell-ell) Specifies that a following d, i, o, u, x, or X conversion specifier applies to a **long long** or
 13150 **unsigned long long** argument; or that a following n conversion specifier applies to a
 13151 pointer to a **long long** argument.

13153 j Specifies that a following d, i, o, u, x, or X conversion specifier applies to an **intmax_t**
 13154 or **uintmax_t** argument; or that a following n conversion specifier applies to a pointer
 13155 to an **intmax_t** argument.

13156 z Specifies that a following d, i, o, u, x, or X conversion specifier applies to a **size_t** or the
 13157 corresponding signed integer type argument; or that a following n conversion specifier
 13158 applies to a pointer to a signed integer type corresponding to a **size_t** argument.

13159 t Specifies that a following d, i, o, u, x, or X conversion specifier applies to a **ptrdiff_t** or
 13160 the corresponding **unsigned** type argument; or that a following n conversion specifier
 13161 applies to a pointer to a **ptrdiff_t** argument.

13162 L Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to a **long**
 13163 **double** argument.

13164 If a length modifier appears with any conversion specifier other than as specified above, the
 13165 behavior is undefined.

13166		The conversion specifiers and their meanings are:
13167	d, i	The int argument shall be converted to a signed decimal in the style "[−]dddd". The
13168		precision specifies the minimum number of digits to appear; if the value being
13169		converted can be represented in fewer digits, it shall be expanded with leading zeros.
13170		The default precision is 1. The result of converting zero with an explicit precision of
13171		zero shall be no characters.
13172	o	The unsigned argument shall be converted to unsigned octal format in the style
13173		"dddd". The precision specifies the minimum number of digits to appear; if the value
13174		being converted can be represented in fewer digits, it shall be expanded with leading
13175		zeros. The default precision is 1. The result of converting zero with an explicit precision
13176		of zero shall be no characters.
13177	u	The unsigned argument shall be converted to unsigned decimal format in the style
13178		"dddd". The precision specifies the minimum number of digits to appear; if the value
13179		being converted can be represented in fewer digits, it shall be expanded with leading
13180		zeros. The default precision is 1. The result of converting zero with an explicit precision
13181		of zero shall be no characters.
13182	x	The unsigned argument shall be converted to unsigned hexadecimal format in the style
13183		"dddd"; the letters "abcdef" are used. The precision specifies the minimum number
13184		of digits to appear; if the value being converted can be represented in fewer digits, it
13185		shall be expanded with leading zeros. The default precision is 1. The result of
13186		converting zero with an explicit precision of zero shall be no characters.
13187	X	Equivalent to the x conversion specifier, except that letters "ABCDEF" are used instead
13188		of "abcdef".
13189	f, F	The double argument shall be converted to decimal notation in the style
13190		"[−]ddd.ddd", where the number of digits after the radix character is equal to the
13191		precision specification. If the precision is missing, it shall be taken as 6; if the precision
13192		is explicitly zero and no '#' flag is present, no radix character shall appear. If a radix
13193		character appears, at least one digit appears before it. The low-order digit shall be
13194		rounded in an implementation-defined manner.
13195		A double argument representing an infinity shall be converted in one of the styles
13196		"[−]inf" or "[−]infinity"; which style is implementation-defined. A double
13197		argument representing a NaN shall be converted in one of the styles "[−]nan(<i>n-char-sequence</i>)"
13198		or "[−]nan"; which style, and the meaning of any <i>n-char-sequence</i> ,
13199		is implementation-defined. The F conversion specifier produces "INF", "INFINITY",
13200		or "NAN" instead of "inf", "infinity", or "nan", respectively.
13201	e, E	The double argument shall be converted in the style "[−]d.ddde±dd", where there is
13202		one digit before the radix character (which is non-zero if the argument is non-zero) and
13203		the number of digits after it is equal to the precision; if the precision is missing, it shall
13204		be taken as 6; if the precision is zero and no '#' flag is present, no radix character shall
13205		appear. The low-order digit shall be rounded in an implementation-defined manner.
13206		The E conversion specifier shall produce a number with 'E' instead of 'e'
13207		introducing the exponent. The exponent shall always contain at least two digits. If the
13208		value is zero, the exponent shall be zero.
13209		A double argument representing an infinity or NaN shall be converted in the style of
13210		an f or F conversion specifier.
13211	g, G	The double argument shall be converted in the style f or e (or in the style F or E in the
13212		case of a G conversion specifier), with the precision specifying the number of significant

13213		digits. If an explicit precision is zero, it shall be taken as 1. The style used depends on
13214		the value converted; style <code>e</code> (or <code>E</code>) shall be used only if the exponent resulting from
13215		such a conversion is less than <code>-4</code> or greater than or equal to the precision. Trailing zeros
13216		shall be removed from the fractional portion of the result; a radix character shall appear
13217		only if it is followed by a digit or a <code>'#'</code> flag is present.
13218		A double argument representing an infinity or NaN shall be converted in the style of
13219		an <code>f</code> or <code>F</code> conversion specifier.
13220	a, A	A double argument representing a floating-point number shall be converted in the
13221		style <code>"[-]0xh.hhhhp±d"</code> , where there is one hexadecimal digit (which shall be non-
13222		zero if the argument is a normalized floating-point number and is otherwise
13223		unspecified) before the decimal-point character and the number of hexadecimal digits
13224		after it is equal to the precision; if the precision is missing and <code>FLT_RADIX</code> is a power
13225		of 2, then the precision shall be sufficient for an exact representation of the value; if the
13226		precision is missing and <code>FLT_RADIX</code> is not a power of 2, then the precision shall be
13227		sufficient to distinguish values of type double , except that trailing zeros may be
13228		omitted; if the precision is zero and the <code>'#'</code> flag is not specified, no decimal-point
13229		character shall appear. The letters <code>"abcdef"</code> shall be used for a conversion and the
13230		letters <code>"ABCDEF"</code> for A conversion. The A conversion specifier produces a number with
13231		<code>'X'</code> and <code>'P'</code> instead of <code>'x'</code> and <code>'p'</code> . The exponent shall always contain at least one
13232		digit, and only as many more digits as necessary to represent the decimal exponent of
13233		2. If the value is zero, the exponent shall be zero.
13234		A double argument representing an infinity or NaN shall be converted in the style of
13235		an <code>f</code> or <code>F</code> conversion specifier.
13236	c	The int argument shall be converted to an unsigned char , and the resulting byte shall
13237		be written.
13238		If an <code>l</code> (ell) qualifier is present, the wint_t argument shall be converted as if by an <code>ls</code>
13239		conversion specification with no precision and an argument that points to a two-
13240		element array of type wchar_t , the first element of which contains the wint_t argument
13241		to the <code>ls</code> conversion specification and the second element contains a null wide
13242		character.
13243	s	The argument shall be a pointer to an array of char . Bytes from the array shall be
13244		written up to (but not including) any terminating null byte. If the precision is specified,
13245		no more than that many bytes shall be written. If the precision is not specified or is
13246		greater than the size of the array, the application shall ensure that the array contains a
13247		null byte.
13248		If an <code>l</code> (ell) qualifier is present, the argument shall be a pointer to an array of type
13249		wchar_t . Wide characters from the array shall be converted to characters (each as if by
13250		a call to the <code>wcrtomb()</code> function, with the conversion state described by an mbstate_t
13251		object initialized to zero before the first wide character is converted) up to and
13252		including a terminating null wide character. The resulting characters shall be written
13253		up to (but not including) the terminating null character (byte). If no precision is
13254		specified, the application shall ensure that the array contains a null wide character. If a
13255		precision is specified, no more than that many characters (bytes) shall be written
13256		(including shift sequences, if any), and the array shall contain a null wide character if,
13257		to equal the character sequence length given by the precision, the function would need
13258		to access a wide character one past the end of the array. In no case shall a partial
13259		character be written.

13260	p	The argument shall be a pointer to void . The value of the pointer is converted to a
13261		sequence of printable characters, in an implementation-defined manner.
13262	n	The argument shall be a pointer to an integer into which is written the number of bytes
13263		written to the output so far by this call to one of the <i>fprintf()</i> functions. No argument is
13264		converted.
13265 XSI	C	Equivalent to <code>lc</code> .
13266 XSI	S	Equivalent to <code>ls</code> .
13267	%	Print a ' % ' character; no argument is converted. The complete conversion specification
13268		shall be <code>%%</code> .
13269		If a conversion specification does not match one of the above forms, the behavior is undefined. If
13270		any argument is not the correct type for the corresponding conversion specification, the
13271		behavior is undefined.
13272		In no case shall a nonexistent or small field width cause truncation of a field; if the result of a
13273		conversion is wider than the field width, the field shall be expanded to contain the conversion
13274		result. Characters generated by <i>fprintf()</i> and <i>printf()</i> are printed as if <i>fputc()</i> had been called.
13275		For the <code>a</code> and <code>A</code> conversion specifiers, if <code>FLT_RADIX</code> is a power of 2, the value shall be correctly
13276		rounded to a hexadecimal floating number with the given precision.
13277		For <code>a</code> and <code>A</code> conversions, if <code>FLT_RADIX</code> is not a power of 2 and the result is not exactly
13278		representable in the given precision, the result should be one of the two adjacent numbers in
13279		hexadecimal floating style with the given precision, with the extra stipulation that the error
13280		should have a correct sign for the current rounding direction.
13281		For the <code>e</code> , <code>E</code> , <code>f</code> , <code>F</code> , <code>g</code> , and <code>G</code> conversion specifiers, if the number of significant decimal digits is at
13282		most <code>DECIMAL_DIG</code> , then the result should be correctly rounded. If the number of significant
13283		decimal digits is more than <code>DECIMAL_DIG</code> but the source value is exactly representable with
13284		<code>DECIMAL_DIG</code> digits, then the result should be an exact representation with trailing zeros.
13285		Otherwise, the source value is bounded by two adjacent decimal strings $L < U$, both having
13286		<code>DECIMAL_DIG</code> significant digits; the value of the resultant decimal string D should satisfy $L \leq$
13287		$D \leq U$, with the extra stipulation that the error should have a correct sign for the current
13288		rounding direction.
13289 CX		The <i>st_ctime</i> and <i>st_mtime</i> fields of the file shall be marked for update between the call to a
13290		successful execution of <i>fprintf()</i> or <i>printf()</i> and the next successful completion of a call to <i>fflush()</i>
13291		or <i>fclose()</i> on the same stream or a call to <i>exit()</i> or <i>abort()</i> .
13292	RETURN VALUE	
13293		Upon successful completion, the <i>fprintf()</i> and <i>printf()</i> functions shall return the number of bytes
13294		transmitted.
13295		Upon successful completion, the <i>sprintf()</i> function shall return the number of bytes written to <i>s</i> ,
13296		excluding the terminating null byte.
13297		Upon successful completion, the <i>snprintf()</i> function shall return the number of bytes that would
13298		be written to <i>s</i> had <i>n</i> been sufficiently large excluding the terminating null byte.
13299		If an output error was encountered, these functions shall return a negative value.
13300		If the value of <i>n</i> is zero on a call to <i>snprintf()</i> , nothing shall be written, the number of bytes that
13301		would have been written had <i>n</i> been sufficiently large excluding the terminating null shall be
13302		returned, and <i>s</i> may be a null pointer.

13303 ERRORS

13304 For the conditions under which *fprintf()* and *printf()* fail and may fail, refer to *fputc()* or
13305 *fputwc()*.

13306 In addition, all forms of *fprintf()* may fail if:

13307 XSI [EILSEQ] A wide-character code that does not correspond to a valid character has been
13308 detected.

13309 XSI [EINVAL] There are insufficient arguments.

13310 The *printf()* and *fprintf()* functions may fail if:

13311 XSI [ENOMEM] Insufficient storage space is available.

13312 The *snprintf()* function shall fail if:

13313 XSI [EOVERFLOW] The value of *n* is greater than {INT_MAX} or the number of bytes needed to
13314 hold the output excluding the terminating null is greater than {INT_MAX}.

13315 EXAMPLES

13316 Printing Language-Independent Date and Time

13317 The following statement can be used to print date and time using a language-independent
13318 format:

13319 `printf(format, weekday, month, day, hour, min);`

13320 For American usage, *format* could be a pointer to the following string:

13321 `"%s, %s %d, %d:%.2d\n"`

13322 This example would produce the following message:

13323 `Sunday, July 3, 10:02`

13324 For German usage, *format* could be a pointer to the following string:

13325 `"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"`

13326 This definition of *format* would produce the following message:

13327 `Sonntag, 3. Juli, 10:02`

13328 Printing File Information

13329 The following example prints information about the type, permissions, and number of links of a
13330 specific file in a directory.

13331 The first two calls to *printf()* use data decoded from a previous *stat()* call. The user-defined
13332 *strperm()* function shall return a string similar to the one at the beginning of the output for the
13333 following command:

13334 `ls -l`

13335 The next call to *printf()* outputs the owner's name if it is found using *getpwuid()*; the *getpwuid()*
13336 function shall return a **passwd** structure from which the name of the user is extracted. If the user
13337 name is not found, the program instead prints out the numeric value of the user ID.

13338 The next call prints out the group name if it is found using *getgrgid()*; *getgrgid()* is very similar to
13339 *getpwuid()* except that it shall return group information based on the group number. Once
13340 again, if the group is not found, the program prints the numeric value of the group for the entry.


```

13341      The final call to printf() prints the size of the file.
13342      #include <stdio.h>
13343      #include <sys/types.h>
13344      #include <pwd.h>
13345      #include <grp.h>
13346
13347      char *strperm (mode_t);
13348      ...
13349      struct stat statbuf;
13350      struct passwd *pwd;
13351      struct group *grp;
13352      ...
13353      printf("%10.10s", strperm (statbuf.st_mode));
13354      printf("%4d", statbuf.st_nlink);
13355
13356      if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
13357          printf(" %-8.8s", pwd->pw_name);
13358      else
13359          printf(" %-8ld", (long) statbuf.st_uid);
13360
13361      if ((grp = getgrgid(statbuf.st_gid)) != NULL)
13362          printf(" %-8.8s", grp->gr_name);
13363      else
13364          printf(" %-8ld", (long) statbuf.st_gid);
13365
13366      printf("%9jd", (intmax_t) statbuf.st_size);
13367      ...

```

Printing a Localized Date String

```

13368      The following example gets a localized date string. The nl_langinfo() function shall return the
13369      localized date string, which specifies the order and layout of the date. The strftime() function
13370      takes this information and, using the tm structure for values, places the date and time
13371      information into datestring. The printf() function then outputs datestring and the name of the
13372      entry.
13373
13374      #include <stdio.h>
13375      #include <time.h>
13376      #include <langinfo.h>
13377      ...
13378      struct dirent *dp;
13379      struct tm *tm;
13380      char datestring[256];
13381      ...
13382      strftime(datestring, sizeof(datestring), nl_langinfo (D_T_FMT), tm);
13383
13384      printf(" %s %s\n", datestring, dp->d_name);
13385      ...

```


Printing Error Information

The following example uses *fprintf()* to write error information to standard error.

In the first group of calls, the program tries to open the password lock file named **LOCKFILE**. If the file already exists, this is an error, as indicated by the **O_EXCL** flag on the *open()* function. If the call fails, the program assumes that someone else is updating the password file, and the program exits.

The next group of calls saves a new password file as the current password file by creating a link between **LOCKFILE** and the new password file **PASSWDFILE**.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#define LOCKFILE "/etc/ptmp"
#define PASSWDFILE "/etc/passwd"
...
int pfd;
...
if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
    exit(1);
}
...
if (link(LOCKFILE, PASSWDFILE) == -1) {
    fprintf(stderr, "Link error: %s\n", strerror(errno));
    exit(1);
}
...
```

Printing Usage Information

The following example checks to make sure the program has the necessary arguments, and uses *fprintf()* to print usage information if the expected number of arguments is not present.

```
#include <stdio.h>
#include <stdlib.h>
...
char *Options = "hdbtl";
...
if (argc < 2) {
    fprintf(stderr, "Usage: %s -%s <file\n", argv[0], Options); exit(1);
}
...
```


Formatting a Decimal String

The following example prints a key and data pair on *stdout*. Note use of the '*' (asterisk) in the format string; this ensures the correct number of decimal places for the element based on the number of elements requested.

```
#include <stdio.h>
...
long i;
char *keyststr;
int elementlen, len;
...
while (len < elementlen) {
...
    printf("%s Element%0*ld\n", keyststr, elementlen, i);
...
}
```

Creating a Filename

The following example creates a filename using information from a previous *getpwnam()* function that returned the HOME directory of the user.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
...
char filename[PATH_MAX+1];
struct passwd *pw;
...
sprintf(filename, "%s/%d.out", pw->pw_dir, getpid());
...
```

Reporting an Event

The following example loops until an event has timed out. The *pause()* function waits forever unless it receives a signal. The *fprintf()* statement should never occur due to the possible return values of *pause()*.

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
...
while (!event_complete) {
...
    if (pause() != -1 || errno != EINTR)
        fprintf(stderr, "pause: unknown error: %s\n", strerror(errno));
}
...
```


Printing Monetary Information

The following example uses *strfmon()* to convert a number and store it as a formatted monetary string named *convbuf*. If the first number is printed, the program prints the format and the description; otherwise, it just prints the number.

```
#include <monetary.h>
#include <stdio.h>
...
struct tblfmt {
    char *format;
    char *description;
};

struct tblfmt table[] = {
    { "%n", "default formatting" },
    { "%11n", "right align within an 11 character field" },
    { "%#5n", "aligned columns for values up to 99999" },
    { "%=*#5n", "specify a fill character" },
    { "%=0#5n", "fill characters do not use grouping" },
    { "%^#5n", "disable the grouping separator" },
    { "%^#5.0n", "round off to whole units" },
    { "%^#5.4n", "increase the precision" },
    { "%(#5n", "use an alternative pos/neg style" },
    { "%!(#5n", "disable the currency symbol" },
};
...
float input[3];
int i, j;
char convbuf[100];
...
strfmon(convbuf, sizeof(convbuf), table[i].format, input[j]);

if (j == 0) {
    printf("%s%s\n", table[i].format,
           convbuf, table[i].description);
}
else {
    printf("%s\n", convbuf);
}
...
```

Printing Wide Characters

The following example prints a series of wide characters. Suppose that "L'@'" expands to three bytes:

```
wchar_t wz [3] = L"@@";           // Zero-terminated
wchar_t wn [3] = L"@@";           // Unterminated

fprintf (stdout, "%ls", wz);       // Outputs 6 bytes
fprintf (stdout, "%ls", wn);       // Undefined because wn has no terminator
fprintf (stdout, "%4ls", wz);      // Outputs 3 bytes
fprintf (stdout, "%4ls", wn);      // Outputs 3 bytes; no terminator needed
fprintf (stdout, "%9ls", wz);      // Outputs 6 bytes
```



```

13515     fprintf (stdout,"%9ls", wn); // Outputs 9 bytes; no terminator needed
13516     fprintf (stdout,"%10ls", wz); // Outputs 6 bytes
13517     fprintf (stdout,"%10ls", wn); // Undefined because wn has no terminator

```

13518 In the last line of the example, after processing three characters, nine bytes have been output.
13519 The fourth character must then be examined to determine whether it converts to one byte or
13520 more. If it converts to more than one byte, the output is only nine bytes. Since there is no fourth
13521 character in the array, the behavior is undefined.

13522 APPLICATION USAGE

13523 If the application calling *fprintf()* has any objects of type **wint_t** or **wchar_t**, it must also include
13524 the **<wchar.h>** header to have these objects defined.

13525 RATIONALE

13526 None.

13527 FUTURE DIRECTIONS

13528 None.

13529 SEE ALSO

13530 *fputc()*, *fscanf()*, *setlocale()*, *strfmon()*, *wcrtomb()*, the Base Definitions volume of
13531 IEEE Std 1003.1-2001, Chapter 7, Locale, **<stdio.h>**, **<wchar.h>**

13532 CHANGE HISTORY

13533 First released in Issue 1. Derived from Issue 1 of the SVID.

13534 Issue 5

13535 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the **l** (ell) qualifier can
13536 now be used with **c** and **s** conversion specifiers.

13537 The *snprintf()* function is new in Issue 5.

13538 Issue 6

13539 Extensions beyond the ISO C standard are marked.

13540 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

13541 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 13542 • The prototypes for *fprintf()*, *printf()*, *snprintf()*, and *sprintf()* are updated, and the XSI
13543 shading is removed from *snprintf()*.
- 13544 • The description of *snprintf()* is aligned with the ISO C standard. Note that this supersedes
13545 the *snprintf()* description in The Open Group Base Resolution bwg98-006, which changed the
13546 behavior from Issue 5.
- 13547 • The DESCRIPTION is updated.

13548 The DESCRIPTION is updated to use the terms “conversion specifier” and “conversion
13549 specification” consistently.

13550 ISO/IEC 9899:1999 standard, Technical Corrigendum No. 1 is incorporated.

13551 An example of printing wide characters is added.

13552 NAME

13553 fputc — put a byte on a stream

13554 SYNOPSIS

13555 #include <stdio.h>

13556 int fputc(int *c*, FILE **stream*);

13557 DESCRIPTION

13558 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 13559 conflict between the requirements described here and the ISO C standard is unintentional. This
 13560 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

13561 The *fputc()* function shall write the byte specified by *c* (converted to an **unsigned char**) to the
 13562 output stream pointed to by *stream*, at the position indicated by the associated file-position
 13563 indicator for the stream (if defined), and shall advance the indicator appropriately. If the file
 13564 cannot support positioning requests, or if the stream was opened with append mode, the byte
 13565 shall be appended to the output stream.

13566 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful
 13567 execution of *fputc()* and the next successful completion of a call to *fflush()* or *fclose()* on the same
 13568 stream or a call to *exit()* or *abort()*.

13569 RETURN VALUE

13570 Upon successful completion, *fputc()* shall return the value it has written. Otherwise, it shall
 13571 CX return EOF, the error indicator for the stream shall be set, and *errno* shall be set to indicate the
 13572 error.

13573 ERRORS

13574 The *fputc()* function shall fail if either the *stream* is unbuffered or the *stream*'s buffer needs to be
 13575 flushed, and:

13576 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the
 13577 process would be delayed in the write operation.

13578 CX [EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for
 13579 writing.

13580 CX [EFBIG] An attempt was made to write to a file that exceeds the maximum file size.

13581 XSI [EFBIG] An attempt was made to write to a file that exceeds the process' file size limit.

13582 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
 13583 offset maximum.

13584 CX [EINTR] The write operation was terminated due to the receipt of a signal, and no data
 13585 was transferred.

13586 CX [EIO] A physical I/O error has occurred, or the process is a member of a
 13587 background process group attempting to write to its controlling terminal,
 13588 TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the
 13589 process group of the process is orphaned. This error may also be returned
 13590 under implementation-defined conditions.

13591 CX [ENOSPC] There was no free space remaining on the device containing the file.

13592 CX [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by
 13593 any process. A SIGPIPE signal shall also be sent to the thread.

13594 The *fputc()* function may fail if:

13595	CX	[ENOMEM]	Insufficient storage space is available.
13596	CX	[ENXIO]	A request was made of a nonexistent device, or the request was outside the capabilities of the device.
13597			
13598	EXAMPLES		
13599			None.
13600	APPLICATION USAGE		
13601			None.
13602	RATIONALE		
13603			None.
13604	FUTURE DIRECTIONS		
13605			None.
13606	SEE ALSO		
13607			<i>ferror()</i> , <i>fopen()</i> , <i>getrlimit()</i> , <i>putc()</i> , <i>puts()</i> , <i>setbuf()</i> , <i>ulimit()</i> , the Base Definitions volume of
13608			IEEE Std 1003.1-2001, <stdio.h>
13609	CHANGE HISTORY		
13610			First released in Issue 1. Derived from Issue 1 of the SVID.
13611	Issue 5		
13612			Large File Summit extensions are added.
13613	Issue 6		
13614			Extensions beyond the ISO C standard are marked.
13615			The following new requirements on POSIX implementations derive from alignment with the
13616			Single UNIX Specification:
13617			<ul style="list-style-type: none"> • The [EIO] and [EFBIG] mandatory error conditions are added.
13618			<ul style="list-style-type: none"> • The [ENOMEM] and [ENXIO] optional error conditions are added.

13619 **NAME**

13620 fputs — put a string on a stream

13621 **SYNOPSIS**

13622 #include <stdio.h>

13623 int fputs(const char *restrict s, FILE *restrict stream);

13624 **DESCRIPTION**

13625 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 13626 conflict between the requirements described here and the ISO C standard is unintentional. This
 13627 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

13628 The *fputs()* function shall write the null-terminated string pointed to by *s* to the stream pointed
 13629 to by *stream*. The terminating null byte shall not be written.

13630 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful
 13631 execution of *fputs()* and the next successful completion of a call to *fflush()* or *fclose()* on the same
 13632 stream or a call to *exit()* or *abort()*.

13633 **RETURN VALUE**

13634 Upon successful completion, *fputs()* shall return a non-negative number. Otherwise, it shall
 13635 CX return EOF, set an error indicator for the stream, and set *errno* to indicate the error.

13636 **ERRORS**13637 Refer to *fputc()*.13638 **EXAMPLES**13639 **Printing to Standard Output**

13640 The following example gets the current time, converts it to a string using *localtime()* and
 13641 *asctime()*, and prints it to standard output using *fputs()*. It then prints the number of minutes to
 13642 an event for which it is waiting.

```
13643 #include <time.h>
13644 #include <stdio.h>
13645 ...
13646 time_t now;
13647 int minutes_to_event;
13648 ...
13649 time(&now);
13650 printf("The time is ");
13651 fputs(asctime(localtime(&now)), stdout);
13652 printf("There are still %d minutes to the event.\n",
13653        minutes_to_event);
13654 ...
```

13655 **APPLICATION USAGE**13656 The *puts()* function appends a <newline> while *fputs()* does not.13657 **RATIONALE**

13658 None.

13659 **FUTURE DIRECTIONS**

13660 None.

13661 **SEE ALSO**

13662 *fopen()*, *putc()*, *puts()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

13663 **CHANGE HISTORY**

13664 First released in Issue 1. Derived from Issue 1 of the SVID.

13665 **Issue 6**

13666 Extensions beyond the ISO C standard are marked.

13667 The *fputs()* prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.

13668 **NAME**

13669 fputcw — put a wide-character code on a stream

13670 **SYNOPSIS**

13671 #include <stdio.h>

13672 #include <wchar.h>

13673 wint_t fputcw(wchar_t *wc*, FILE **stream*);13674 **DESCRIPTION**

13675 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 13676 conflict between the requirements described here and the ISO C standard is unintentional. This
 13677 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

13678 The *fputcw()* function shall write the character corresponding to the wide-character code *wc* to
 13679 the output stream pointed to by *stream*, at the position indicated by the associated file-position
 13680 indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot
 13681 support positioning requests, or if the stream was opened with append mode, the character is
 13682 appended to the output stream. If an error occurs while writing the character, the shift state of
 13683 the output file is left in an undefined state.

13684 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful
 13685 execution of *fputcw()* and the next successful completion of a call to *fflush()* or *fclose()* on the
 13686 same stream or a call to *exit()* or *abort()*.

13687 **RETURN VALUE**

13688 Upon successful completion, *fputcw()* shall return *wc*. Otherwise, it shall return WEOF, the error
 13689 CX indicator for the stream shall be set, and *errno* shall be set to indicate the error.

13690 **ERRORS**

13691 The *fputcw()* function shall fail if either the stream is unbuffered or data in the *stream*'s buffer
 13692 needs to be written, and:

13693 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the
 13694 process would be delayed in the write operation.

13695 CX [EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for
 13696 writing.

13697 CX [EFBIG] An attempt was made to write to a file that exceeds the maximum file size or
 13698 the process' file size limit.

13699 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
 13700 offset maximum associated with the corresponding stream.

13701 [EILSEQ] The wide-character code *wc* does not correspond to a valid character.

13702 CX [EINTR] The write operation was terminated due to the receipt of a signal, and no data
 13703 was transferred.

13704 CX [EIO] A physical I/O error has occurred, or the process is a member of a
 13705 background process group attempting to write to its controlling terminal,
 13706 TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the
 13707 process group of the process is orphaned. This error may also be returned
 13708 under implementation-defined conditions.

13709 CX [ENOSPC] There was no free space remaining on the device containing the file.

13710 CX [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by
 13711 any process. A SIGPIPE signal shall also be sent to the thread.

13712 The *fputwc()* function may fail if:

13713 CX [ENOMEM] Insufficient storage space is available.

13714 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
13715 capabilities of the device.

13716 EXAMPLES

13717 None.

13718 APPLICATION USAGE

13719 None.

13720 RATIONALE

13721 None.

13722 FUTURE DIRECTIONS

13723 None.

13724 SEE ALSO

13725 *ferror()*, *fopen()*, *setbuf()*, *ulimit()*, the Base Definitions volume of IEEE Std 1003.1-2001,
13726 *<stdio.h>*, *<wchar.h>*

13727 CHANGE HISTORY

13728 First released in Issue 4. Derived from the MSE working draft.

13729 Issue 5

13730 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of argument *wc*
13731 is changed from **wint_t** to **wchar_t**.

13732 The Optional Header (OH) marking is removed from **<stdio.h>**.

13733 Large File Summit extensions are added.

13734 Issue 6

13735 Extensions beyond the ISO C standard are marked.

13736 The following new requirements on POSIX implementations derive from alignment with the
13737 Single UNIX Specification:

- 13738 • The [EFBIG] and [EIO] mandatory error conditions are added.
- 13739 • The [ENOMEM] and [ENXIO] optional error conditions are added.

13740 **NAME**

13741 fputws — put a wide-character string on a stream

13742 **SYNOPSIS**

13743 #include <stdio.h>

13744 #include <wchar.h>

13745 int fputws(const wchar_t *restrict ws, FILE *restrict stream);

13746 **DESCRIPTION**

13747 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 13748 conflict between the requirements described here and the ISO C standard is unintentional. This
 13749 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

13750 The *fputws()* function shall write a character string corresponding to the (null-terminated)
 13751 wide-character string pointed to by *ws* to the stream pointed to by *stream*. No character
 13752 corresponding to the terminating null wide-character code shall be written.

13753 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful
 13754 execution of *fputws()* and the next successful completion of a call to *fflush()* or *fclose()* on the
 13755 same stream or a call to *exit()* or *abort()*.

13756 **RETURN VALUE**

13757 Upon successful completion, *fputws()* shall return a non-negative number. Otherwise, it shall
 13758 CX return -1, set an error indicator for the stream, and set *errno* to indicate the error.

13759 **ERRORS**13760 Refer to *fputwc()*.13761 **EXAMPLES**

13762 None.

13763 **APPLICATION USAGE**13764 The *fputws()* function does not append a <newline>.13765 **RATIONALE**

13766 None.

13767 **FUTURE DIRECTIONS**

13768 None.

13769 **SEE ALSO**13770 *fopen()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>, <wchar.h>13771 **CHANGE HISTORY**

13772 First released in Issue 4. Derived from the MSE working draft.

13773 **Issue 5**

13774 The Optional Header (OH) marking is removed from <stdio.h>.

13775 **Issue 6**

13776 Extensions beyond the ISO C standard are marked.

13777 The *fputws()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

13778 NAME

13779 fread — binary input

13780 SYNOPSIS

13781 #include <stdio.h>

```
13782 size_t fread(void *restrict ptr, size_t size, size_t nitems,
13783 FILE *restrict stream);
```

13784 DESCRIPTION

13785 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 13786 conflict between the requirements described here and the ISO C standard is unintentional. This
 13787 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

13788 The *fread()* function shall read into the array pointed to by *ptr* up to *nitems* elements whose size
 13789 is specified by *size* in bytes, from the stream pointed to by *stream*. For each object, *size* calls shall
 13790 be made to the *fgetc()* function and the results stored, in the order read, in an array of **unsigned**
 13791 **char** exactly overlaying the object. The file position indicator for the stream (if defined) shall be
 13792 advanced by the number of bytes successfully read. If an error occurs, the resulting value of the
 13793 file position indicator for the stream is unspecified. If a partial element is read, its value is
 13794 unspecified.

13795 CX The *fread()* function may mark the *st_atime* field of the file associated with *stream* for update. The
 13796 *st_atime* field shall be marked for update by the first successful execution of *fgetc()*, *fgets()*,
 13797 *fgetwc()*, *fgetws()*, *fread()*, *fscanf()*, *getc()*, *getchar()*, *gets()*, or *scanf()* using *stream* that returns
 13798 data not supplied by a prior call to *ungetc()* or *ungetwc()*.

13799 RETURN VALUE

13800 Upon successful completion, *fread()* shall return the number of elements successfully read which
 13801 is less than *nitems* only if a read error or end-of-file is encountered. If *size* or *nitems* is 0, *fread()*
 13802 shall return 0 and the contents of the array and the state of the stream remain unchanged.
 13803 CX Otherwise, if a read error occurs, the error indicator for the stream shall be set, and *errno* shall be
 13804 set to indicate the error.

13805 ERRORS

13806 Refer to *fgetc()*.

13807 EXAMPLES

13808 Reading from a Stream

13809 The following example reads a single element from the *fp* stream into the array pointed to by *buf*.

```
13810 #include <stdio.h>
13811 ...
13812 size_t bytes_read;
13813 char buf[100];
13814 FILE *fp;
13815 ...
13816 bytes_read = fread(buf, sizeof(buf), 1, fp);
13817 ...
```

13818 APPLICATION USAGE

13819 The *ferror()* or *feof()* functions must be used to distinguish between an error condition and an
 13820 end-of-file condition.

13821 Because of possible differences in element length and byte ordering, files written using *fwrite()*
 13822 are application-dependent, and possibly cannot be read using *fread()* by a different application

13823 or by the same application on a different processor.

13824 **RATIONALE**

13825 None.

13826 **FUTURE DIRECTIONS**

13827 None.

13828 **SEE ALSO**

13829 *feof()*, *ferror()*, *fgetc()*, *fopen()*, *getc()*, *gets()*, *scanf()*, the Base Definitions volume of
13830 IEEE Std 1003.1-2001, <**stdio.h**>

13831 **CHANGE HISTORY**

13832 First released in Issue 1. Derived from Issue 1 of the SVID.

13833 **Issue 6**

13834 Extensions beyond the ISO C standard are marked.

13835 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 13836 • The *fread()* prototype is updated.
- 13837 • The DESCRIPTION is updated to describe how the bytes from a call to *fgetc()* are stored.

13838 **NAME**

13839 free — free allocated memory

13840 **SYNOPSIS**

13841 #include <stdlib.h>

13842 void free(void *ptr);

13843 **DESCRIPTION**

13844 CX The functionality described on this reference page is aligned with the ISO C standard. Any
13845 conflict between the requirements described here and the ISO C standard is unintentional. This
13846 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

13847 The *free()* function shall cause the space pointed to by *ptr* to be deallocated; that is, made
13848 available for further allocation. If *ptr* is a null pointer, no action shall occur. Otherwise, if the
13849 ADV argument does not match a pointer earlier returned by the *calloc()*, *malloc()*, *posix_memalign()*,
13850 XSI *realloc()*, or *strdup()* function, or if the space has been deallocated by a call to *free()* or *realloc()*,
13851 the behavior is undefined.

13852 Any use of a pointer that refers to freed space results in undefined behavior.

13853 **RETURN VALUE**13854 The *free()* function shall not return a value.13855 **ERRORS**

13856 No errors are defined.

13857 **EXAMPLES**

13858 None.

13859 **APPLICATION USAGE**

13860 There is now no requirement for the implementation to support the inclusion of <malloc.h>.

13861 **RATIONALE**

13862 None.

13863 **FUTURE DIRECTIONS**

13864 None.

13865 **SEE ALSO**13866 *calloc()*, *malloc()*, *realloc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>13867 **CHANGE HISTORY**

13868 First released in Issue 1. Derived from Issue 1 of the SVID.

13869 **Issue 6**13870 Reference to the *valloc()* function is removed.

13871 NAME

13872 freeaddrinfo, getaddrinfo — get address information

13873 SYNOPSIS

```

13874 #include <sys/socket.h>
13875 #include <netdb.h>

13876 void freeaddrinfo(struct addrinfo *ai);
13877 int getaddrinfo(const char *restrict nodename,
13878                const char *restrict servname,
13879                const struct addrinfo *restrict hints,
13880                struct addrinfo **restrict res);

```

13881 DESCRIPTION

13882 The *freeaddrinfo()* function shall free one or more **addrinfo** structures returned by *getaddrinfo()*,
 13883 along with any additional storage associated with those structures. If the *ai_next* field of the
 13884 structure is not null, the entire list of structures shall be freed. The *freeaddrinfo()* function shall
 13885 support the freeing of arbitrary sublists of an **addrinfo** list originally returned by *getaddrinfo()*.

13886 The *getaddrinfo()* function shall translate the name of a service location (for example, a host
 13887 name) and/or a service name and shall return a set of socket addresses and associated
 13888 information to be used in creating a socket with which to address the specified service.

13889 The *freeaddrinfo()* and *getaddrinfo()* functions shall be thread-safe.

13890 The *nodename* and *servname* arguments are either null pointers or pointers to null-terminated
 13891 strings. One or both of these two arguments shall be supplied by the application as a non-null
 13892 pointer.

13893 The format of a valid name depends on the address family or families. If a specific family is not
 13894 given and the name could be interpreted as valid within multiple supported families, the
 13895 implementation shall attempt to resolve the name in all supported families and, in absence of
 13896 errors, one or more results shall be returned.

13897 If the *nodename* argument is not null, it can be a descriptive name or can be an address string. If
 13898 IP6 the specified address family is AF_INET, AF_INET6, or AF_UNSPEC, valid descriptive names
 13899 include host names. If the specified address family is AF_INET or AF_UNSPEC, address strings
 13900 using Internet standard dot notation as specified in *inet_addr()* are valid.

13901 IP6 If the specified address family is AF_INET6 or AF_UNSPEC, standard IPv6 text forms described
 13902 in *inet_ntop()* are valid.

13903 If *nodename* is not null, the requested service location is named by *nodename*; otherwise, the
 13904 requested service location is local to the caller.

13905 If *servname* is null, the call shall return network-level addresses for the specified *nodename*. If
 13906 *servname* is not null, it is a null-terminated character string identifying the requested service. This
 13907 can be either a descriptive name or a numeric representation suitable for use with the address
 13908 IP6 family or families. If the specified address family is AF_INET, AF_INET6, or AF_UNSPEC, the
 13909 service can be specified as a string specifying a decimal port number.

13910 If the *hints* argument is not null, it refers to a structure containing input values that may direct
 13911 the operation by providing options and by limiting the returned information to a specific socket
 13912 type, address family, and/or protocol. In this *hints* structure every member other than *ai_flags*,
 13913 *ai_family*, *ai_socktype*, and *ai_protocol* shall be set to zero or a null pointer. A value of
 13914 AF_UNSPEC for *ai_family* means that the caller shall accept any address family. A value of zero
 13915 for *ai_socktype* means that the caller shall accept any socket type. A value of zero for *ai_protocol*
 13916 means that the caller shall accept any protocol. If *hints* is a null pointer, the behavior shall be as if

13917 it referred to a structure containing the value zero for the *ai_flags*, *ai_socktype*, and *ai_protocol*
 13918 fields, and AF_UNSPEC for the *ai_family* field.

13919 The *ai_flags* field to which the *hints* parameter points shall be set to zero or be the bitwise-
 13920 inclusive OR of one or more of the values AI_PASSIVE, AI_CANONNAME,
 13921 AI_NUMERICHOST, and AI_NUMERICSERV.

13922 If the AI_PASSIVE flag is specified, the returned address information shall be suitable for use in
 13923 binding a socket for accepting incoming connections for the specified service. In this case, if the
 13924 *nodename* argument is null, then the IP address portion of the socket address structure shall be
 13925 set to INADDR_ANY for an IPv4 address or IN6ADDR_ANY_INIT for an IPv6 address. If the
 13926 AI_PASSIVE flag is not specified, the returned address information shall be suitable for a call to
 13927 *connect()* (for a connection-mode protocol) or for a call to *connect()*, *sendto()*, or *sendmsg()* (for a
 13928 connectionless protocol). In this case, if the *nodename* argument is null, then the IP address
 13929 portion of the socket address structure shall be set to the loopback address.

13930 If the AI_CANONNAME flag is specified and the *nodename* argument is not null, the function
 13931 shall attempt to determine the canonical name corresponding to *nodename* (for example, if
 13932 *nodename* is an alias or shorthand notation for a complete name).

13933 If the AI_NUMERICHOST flag is specified, then a non-null *nodename* string supplied shall be a
 13934 numeric host address string. Otherwise, an [EAI_NONAME] error is returned. This flag shall
 13935 prevent any type of name resolution service (for example, the DNS) from being invoked.

13936 If the AI_NUMERICSERV flag is specified, then a non-null *servname* string supplied shall be a
 13937 numeric port string. Otherwise, an [EAI_NONAME] error shall be returned. This flag shall
 13938 prevent any type of name resolution service (for example, NIS+) from being invoked.

13939 IP6 If the AI_V4MAPPED flag is specified along with an *ai_family* of AF_INET6, then *getaddrinfo()*
 13940 shall return IPv4-mapped IPv6 addresses on finding no matching IPv6 addresses (*ai_addrlen*
 13941 shall be 16). The AI_V4MAPPED flag shall be ignored unless *ai_family* equals AF_INET6. If the
 13942 AI_ALL flag is used with the AI_V4MAPPED flag, then *getaddrinfo()* shall return all matching
 13943 IPv6 and IPv4 addresses. The AI_ALL flag without the AI_V4MAPPED flag is ignored.

13944 The *ai_socktype* field to which argument *hints* points specifies the socket type for the service, as
 13945 defined in *socket()*. If a specific socket type is not given (for example, a value of zero) and the
 13946 service name could be interpreted as valid with multiple supported socket types, the
 13947 implementation shall attempt to resolve the service name for all supported socket types and, in
 13948 the absence of errors, all possible results shall be returned. A non-zero socket type value shall
 13949 limit the returned information to values with the specified socket type.

13950 If the *ai_family* field to which *hints* points has the value AF_UNSPEC, addresses shall be
 13951 returned for use with any address family that can be used with the specified *nodename* and/or
 13952 *servname*. Otherwise, addresses shall be returned for use only with the specified address family.
 13953 If *ai_family* is not AF_UNSPEC and *ai_protocol* is not zero, then addresses are returned for use
 13954 only with the specified address family and protocol; the value of *ai_protocol* shall be interpreted
 13955 as in a call to the *socket()* function with the corresponding values of *ai_family* and *ai_protocol*.

13956 **RETURN VALUE**

13957 A zero return value for *getaddrinfo()* indicates successful completion; a non-zero return value
 13958 indicates failure. The possible values for the failures are listed in the ERRORS section.

13959 Upon successful return of *getaddrinfo()*, the location to which *res* points shall refer to a linked list
 13960 of **addrinfo** structures, each of which shall specify a socket address and information for use in
 13961 creating a socket with which to use that socket address. The list shall include at least one
 13962 **addrinfo** structure. The *ai_next* field of each structure contains a pointer to the next structure on
 13963 the list, or a null pointer if it is the last structure on the list. Each structure on the list shall

13964 include values for use with a call to the *socket()* function, and a socket address for use with the
 13965 *connect()* function or, if the *AI_PASSIVE* flag was specified, for use with the *bind()* function. The
 13966 fields *ai_family*, *ai_socktype*, and *ai_protocol* shall be usable as the arguments to the *socket()*
 13967 function to create a socket suitable for use with the returned address. The fields *ai_addr* and
 13968 *ai_addrlen* are usable as the arguments to the *connect()* or *bind()* functions with such a socket,
 13969 according to the *AI_PASSIVE* flag.

13970 If *nodename* is not null, and if requested by the *AI_CANONNAME* flag, the *ai_canonname* field of
 13971 the first returned **addrinfo** structure shall point to a null-terminated string containing the
 13972 canonical name corresponding to the input *nodename*; if the canonical name is not available, then
 13973 *ai_canonname* shall refer to the *nodename* argument or a string with the same contents. The
 13974 contents of the *ai_flags* field of the returned structures are undefined.

13975 All fields in socket address structures returned by *getaddrinfo()* that are not filled in through an
 13976 explicit argument (for example, *sin6_flowinfo*) shall be set to zero.

13977 **Note:** This makes it easier to compare socket address structures.

13978 ERRORS

13979 The *getaddrinfo()* function shall fail and return the corresponding value if:

13980 [EAI_AGAIN] The name could not be resolved at this time. Future attempts may succeed.

13981 [EAI_BADFLAGS]

13982 The *flags* parameter had an invalid value.

13983 [EAI_FAIL] A non-recoverable error occurred when attempting to resolve the name.

13984 [EAI_FAMILY] The address family was not recognized.

13985 [EAI_MEMORY] There was a memory allocation failure when trying to allocate storage for the
 13986 return value.

13987 [EAI_NONAME] The name does not resolve for the supplied parameters.

13988 Neither *nodename* nor *servname* were supplied. At least one of these shall be
 13989 supplied.

13990 [EAI_SERVICE] The service passed was not recognized for the specified socket type.

13991 [EAI_SOCKTYPE]

13992 The intended socket type was not recognized.

13993 [EAI_SYSTEM] A system error occurred; the error code can be found in *errno*.

13994 [EAI_OVERFLOW]

13995 An argument buffer overflowed.

13996 EXAMPLES

13997 None.

13998 APPLICATION USAGE

13999 If the caller handles only TCP and not UDP, for example, then the *ai_protocol* member of the *hints*
 14000 structure should be set to *IPPROTO_TCP* when *getaddrinfo()* is called.

14001 If the caller handles only IPv4 and not IPv6, then the *ai_family* member of the *hints* structure
 14002 should be set to *AF_INET* when *getaddrinfo()* is called.

14003 RATIONALE

14004 None.

14005 **FUTURE DIRECTIONS**

14006 None.

14007 **SEE ALSO**

14008 *connect()*, *gai_strerror()*, *gethostbyaddr()*, *getnameinfo()*, *getservbyname()*, *socket()*, the Base
14009 Definitions volume of IEEE Std 1003.1-2001, <netdb.h>, <sys/socket.h>

14010 **CHANGE HISTORY**

14011 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

14012 The **restrict** keyword is added to the *getaddrinfo()* prototype for alignment with the
14013 ISO/IEC 9899:1999 standard.

14014 NAME

14015 freopen — open a stream

14016 SYNOPSIS

14017 #include <stdio.h>

14018 FILE *freopen(const char *restrict filename, const char *restrict mode,
 14019 FILE *restrict stream);

14020 DESCRIPTION

14021 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 14022 conflict between the requirements described here and the ISO C standard is unintentional. This
 14023 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

14024 The *freopen()* function shall first attempt to flush the stream and close any file descriptor
 14025 associated with *stream*. Failure to flush or close the file descriptor successfully shall be ignored.
 14026 The error and end-of-file indicators for the stream shall be cleared.

14027 The *freopen()* function shall open the file whose pathname is the string pointed to by *filename* and
 14028 associate the stream pointed to by *stream* with it. The *mode* argument shall be used just as in
 14029 *fopen()*.

14030 The original stream shall be closed regardless of whether the subsequent open succeeds.

14031 If *filename* is a null pointer, the *freopen()* function shall attempt to change the mode of the stream
 14032 to that specified by *mode*, as if the name of the file currently associated with the stream had been
 14033 used. It is implementation-defined which changes of mode are permitted (if any), and under
 14034 what circumstances.

14035 XSI After a successful call to the *freopen()* function, the orientation of the stream shall be cleared, the
 14036 encoding rule shall be cleared, and the associated **mbstate_t** object shall be set to describe an
 14037 initial conversion state.

14038 CX The largest value that can be represented correctly in an object of type **off_t** shall be established
 14039 as the offset maximum in the open file description.

14040 RETURN VALUE

14041 Upon successful completion, *freopen()* shall return the value of *stream*. Otherwise, a null pointer
 14042 CX shall be returned, and *errno* shall be set to indicate the error.

14043 ERRORS

14044 The *freopen()* function shall fail if:

14045 CX [EACCES] Search permission is denied on a component of the path prefix, or the file
 14046 exists and the permissions specified by *mode* are denied, or the file does not
 14047 exist and write permission is denied for the parent directory of the file to be
 14048 created.

14049 CX [EINTR] A signal was caught during *freopen()*.

14050 CX [EISDIR] The named file is a directory and *mode* requires write access.

14051 CX [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 14052 argument.

14053 CX [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

14054 CX [ENAMETOOLONG] The length of the *filename* argument exceeds {PATH_MAX} or a pathname
 14055 component is longer than {NAME_MAX}.
 14056

14057 CX	[ENFILE]	The maximum allowable number of files is currently open in the system.
14058 CX	[ENOENT]	A component of <i>filename</i> does not name an existing file or <i>filename</i> is an empty string.
14059		
14060 CX	[ENOSPC]	The directory or file system that would contain the new file cannot be expanded, the file does not exist, and it was to be created.
14061		
14062 CX	[ENOTDIR]	A component of the path prefix is not a directory.
14063 CX	[ENXIO]	The named file is a character special or block special file, and the device associated with this special file does not exist.
14064		
14065 CX	[EOVERFLOW]	The named file is a regular file and the size of the file cannot be represented correctly in an object of type <code>off_t</code> .
14066		
14067 CX	[EROFS]	The named file resides on a read-only file system and <i>mode</i> requires write access.
14068		
14069	The <i>freopen()</i> function may fail if:	
14070 CX	[EINVAL]	The value of the <i>mode</i> argument is not valid.
14071 CX	[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the <i>path</i> argument.
14072		
14073 CX	[ENAMETOOLONG]	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.
14074		
14075		
14076 CX	[ENOMEM]	Insufficient storage space is available.
14077 CX	[ENXIO]	A request was made of a nonexistent device, or the request was outside the capabilities of the device.
14078		
14079 CX	[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed and <i>mode</i> requires write access.
14080		

14081 EXAMPLES

14082 Directing Standard Output to a File

14083 The following example logs all standard output to the `/tmp/logfile` file.

```

14084 #include <stdio.h>
14085 ...
14086 FILE *fp;
14087 ...
14088 fp = freopen ("/tmp/logfile", "a+", stdout);
14089 ...

```

14090 APPLICATION USAGE

14091 The *freopen()* function is typically used to attach the preopened *streams* associated with *stdin*,
 14092 *stdout*, and *stderr* to other files.

14093 RATIONALE

14094 None.

14095 **FUTURE DIRECTIONS**

14096 None.

14097 **SEE ALSO**

14098 *fclose()*, *fopen()*, *fdopen()*, *mbsinit()*, the Base Definitions volume of IEEE Std 1003.1-2001,
14099 <stdio.h>

14100 **CHANGE HISTORY**

14101 First released in Issue 1. Derived from Issue 1 of the SVID.

14102 **Issue 5**

14103 The DESCRIPTION is updated to indicate that the orientation of the stream is cleared and the
14104 conversion state of the stream is set to an initial conversion state by a successful call to the
14105 *freopen()* function.

14106 Large File Summit extensions are added.

14107 **Issue 6**

14108 Extensions beyond the ISO C standard are marked.

14109 The following new requirements on POSIX implementations derive from alignment with the
14110 Single UNIX Specification:

- 14111 • In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file
14112 description. This change is to support large files.
- 14113 • In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support
14114 large files.
- 14115 • The [ELOOP] mandatory error condition is added.
- 14116 • A second [ENAMETOOLONG] is added as an optional error condition.
- 14117 • The [EINVAL], [ENOMEM], [ENXIO], and [ETXTBSY] optional error conditions are added.

14118 The following changes are made for alignment with the ISO/IEC 9899: 1999 standard:

- 14119 • The *freopen()* prototype is updated.
- 14120 • The DESCRIPTION is updated.

14121 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
14122 [ELOOP] error condition is added.

14123 The DESCRIPTION is updated regarding failure to close, changing the “file” to “file descriptor”.

14124 **NAME**

14125 frexp, frexpf, frexpl — extract mantissa and exponent from a double precision number

14126 **SYNOPSIS**

14127 #include <math.h>

14128 double frexp(double *num*, int **exp*);

14129 float frexpf(float *num*, int **exp*);

14130 long double frexpl(long double *num*, int **exp*);

14131 **DESCRIPTION**

14132 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 14133 conflict between the requirements described here and the ISO C standard is unintentional. This
 14134 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

14135 These functions shall break a floating-point number *num* into a normalized fraction and an
 14136 integral power of 2. The integer exponent shall be stored in the **int** object pointed to by *exp*.

14137 **RETURN VALUE**

14138 For finite arguments, these functions shall return the value *x*, such that *x* has a magnitude in the
 14139 interval $[1/2, 1)$ or 0, and *num* equals *x* times 2 raised to the power **exp*.

14140 MX If *num* is NaN, a NaN shall be returned, and the value of **exp* is unspecified.

14141 If *num* is ± 0 , ± 0 shall be returned, and the value of **exp* shall be 0.

14142 If *num* is $\pm \text{Inf}$, *num* shall be returned, and the value of **exp* is unspecified.

14143 **ERRORS**

14144 No errors are defined.

14145 **EXAMPLES**

14146 None.

14147 **APPLICATION USAGE**

14148 None.

14149 **RATIONALE**

14150 None.

14151 **FUTURE DIRECTIONS**

14152 None.

14153 **SEE ALSO**

14154 *isnan()*, *ldexp()*, *modf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <math.h>

14155 **CHANGE HISTORY**

14156 First released in Issue 1. Derived from Issue 1 of the SVID.

14157 **Issue 5**

14158 The DESCRIPTION is updated to indicate how an application should check for an error. This
 14159 text was previously published in the APPLICATION USAGE section.

14160 **Issue 6**

14161 The *frexpf()* and *frexpl()* functions are added for alignment with the ISO/IEC 9899:1999
 14162 standard.

14163	The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
14164	revised to align with the ISO/IEC 9899:1999 standard.
14165	IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
14166	marked.

14167 **NAME**

14168 fscanf, scanf, sscanf — convert formatted input

14169 **SYNOPSIS**

14170 #include <stdio.h>

14171 int fscanf(FILE *restrict *stream*, const char *restrict *format*, ...);14172 int scanf(const char *restrict *format*, ...);14173 int sscanf(const char *restrict *s*, const char *restrict *format*, ...);14174 **DESCRIPTION**

14175 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 14176 conflict between the requirements described here and the ISO C standard is unintentional. This
 14177 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

14178 The *fscanf()* function shall read from the named input *stream*. The *scanf()* function shall read
 14179 from the standard input stream *stdin*. The *sscanf()* function shall read from the string *s*. Each
 14180 function reads bytes, interprets them according to a format, and stores the results in its
 14181 arguments. Each expects, as arguments, a control string *format* described below, and a set of
 14182 *pointer* arguments indicating where the converted input should be stored. The result is
 14183 undefined if there are insufficient arguments for the format. If the format is exhausted while
 14184 arguments remain, the excess arguments shall be evaluated but otherwise ignored.

14185 XSI Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than
 14186 to the next unused argument. In this case, the conversion specifier character % (see below) is
 14187 replaced by the sequence "%n\$", where *n* is a decimal integer in the range [1,{NL_ARGMAX}].
 14188 This feature provides for the definition of format strings that select arguments in an order
 14189 appropriate to specific languages. In format strings containing the "%n\$" form of conversion
 14190 specifications, it is unspecified whether numbered arguments in the argument list can be
 14191 referenced from the format string more than once.

14192 The *format* can contain either form of a conversion specification—that is, % or "%n\$"—but the
 14193 two forms cannot be mixed within a single *format* string. The only exception to this is that %% or
 14194 %* can be mixed with the "%n\$" form. When numbered argument specifications are used,
 14195 specifying the *N*th argument requires that all the leading arguments, from the first to the
 14196 (*N*–1)th, are pointers.

14197 CX The *fscanf()* function in all its forms shall allow detection of a language-dependent radix
 14198 character in the input string. The radix character is defined in the program's locale (category
 14199 LC_NUMERIC). In the POSIX locale, or in a locale where the radix character is not defined, the
 14200 radix character shall default to a period ('.').

14201 The format is a character string, beginning and ending in its initial shift state, if any, composed
 14202 of zero or more directives. Each directive is composed of one of the following: one or more
 14203 white-space characters (<space>*s*, <tab>*s*, <newline>*s*, <vertical-tab>*s*, or <form-feed>*s*); an
 14204 ordinary character (neither '%' nor a white-space character); or a conversion specification. Each
 14205 XSI conversion specification is introduced by the character '%' or the character sequence "%n\$",
 14206 after which the following appear in sequence:

- 14207 • An optional assignment-suppressing character '*'.
- 14208 • An optional non-zero decimal integer that specifies the maximum field width.
- 14209 • An option length modifier that specifies the size of the receiving object.
- 14210 • A *conversion specifier* character that specifies the type of conversion to be applied. The valid
 14211 conversion specifiers are described below.

14212 The *fscanf()* functions shall execute each directive of the format in turn. If a directive fails, as
 14213 detailed below, the function shall return. Failures are described as input failures (due to the
 14214 unavailability of input bytes) or matching failures (due to inappropriate input).

14215 A directive composed of one or more white-space characters shall be executed by reading input
 14216 until no more valid input can be read, or up to the first byte which is not a white-space character,
 14217 which remains unread.

14218 A directive that is an ordinary character shall be executed as follows: the next byte shall be read
 14219 from the input and compared with the byte that comprises the directive; if the comparison
 14220 shows that they are not equivalent, the directive shall fail, and the differing and subsequent
 14221 bytes shall remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a
 14222 character from being read, the directive shall fail.

14223 A directive that is a conversion specification defines a set of matching input sequences, as
 14224 described below for each conversion character. A conversion specification shall be executed in
 14225 the following steps.

14226 Input white-space characters (as specified by *isspace()*) shall be skipped, unless the conversion
 14227 specification includes a *[*, *c*, *C*, or *n* conversion specifier.

14228 An item shall be read from the input, unless the conversion specification includes an *n*
 14229 conversion specifier. An input item shall be defined as the longest sequence of input bytes (up to
 14230 any specified maximum field width, which may be measured in characters or bytes dependent
 14231 on the conversion specifier) which is an initial subsequence of a matching sequence. The first
 14232 byte, if any, after the input item shall remain unread. If the length of the input item is 0, the
 14233 execution of the conversion specification shall fail; this condition is a matching failure, unless
 14234 end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is
 14235 an input failure.

14236 Except in the case of a *%* conversion specifier, the input item (or, in the case of a *%n* conversion
 14237 specification, the count of input bytes) shall be converted to a type appropriate to the conversion
 14238 character. If the input item is not a matching sequence, the execution of the conversion
 14239 specification fails; this condition is a matching failure. Unless assignment suppression was
 14240 indicated by a *'*'*, the result of the conversion shall be placed in the object pointed to by the
 14241 first argument following the *format* argument that has not already received a conversion result if
 14242 XSI the conversion specification is introduced by *%*, or in the *n*th argument if introduced by the
 14243 character sequence *"%n\$"*. If this object does not have an appropriate type, or if the result of the
 14244 conversion cannot be represented in the space provided, the behavior is undefined.

14245 The length modifiers and their meanings are:

14246 hh Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an
 14247 argument with type pointer to **signed char** or **unsigned char**.

14248 h Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an
 14249 argument with type pointer to **short** or **unsigned short**.

14250 l (ell) Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an
 14251 argument with type pointer to **long** or **unsigned long**; that a following *a*, *A*, *e*, *E*, *f*, *F*, *g*,
 14252 or *G* conversion specifier applies to an argument with type pointer to **double**; or that a
 14253 following *c*, *s*, or *[* conversion specifier applies to an argument with type pointer to
 14254 **wchar_t**.

14255 ll (ell-ell) Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an
 14256 argument with type pointer to **long long** or **unsigned long long**.
 14257

14258	j	Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to intmax_t or uintmax_t .
14259		
14260	z	Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to size_t or the corresponding signed integer type.
14261		
14262	t	Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to ptrdiff_t or the corresponding unsigned type.
14263		
14264	L	Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to long double .
14265		
14266		If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.
14267		
14268		The following conversion specifiers are valid:
14269	d	Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of <i>strtol()</i> with the value 10 for the <i>base</i> argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to int .
14270		
14271		
14272		
14273	i	Matches an optionally signed integer, whose format is the same as expected for the subject sequence of <i>strtol()</i> with 0 for the <i>base</i> argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to int .
14274		
14275		
14276		
14277	o	Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of <i>strtoul()</i> with the value 8 for the <i>base</i> argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to unsigned .
14278		
14279		
14280		
14281	u	Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of <i>strtoul()</i> with the value 10 for the <i>base</i> argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to unsigned .
14282		
14283		
14284		
14285	x	Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of <i>strtoul()</i> with the value 16 for the <i>base</i> argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to unsigned .
14286		
14287		
14288		
14289	a, e, f, g	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected for the subject sequence of <i>strtod()</i> . In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to float .
14290		
14291		
14292		
14293		
14294		If the <i>fprintf()</i> family of functions generates character string representations for infinity and NaN (a symbolic entity encoded in floating-point format) to support IEEE Std 754-1985, the <i>fscanf()</i> family of functions shall recognize them as input.
14295		
14296		
14297	s	Matches a sequence of bytes that are not white-space characters. The application shall ensure that the corresponding argument is a pointer to the initial byte of an array of char , signed char , or unsigned char large enough to accept the sequence and a terminating null character code, which shall be added automatically.
14298		
14299		
14300		
14301		If an l (ell) qualifier is present, the input is a sequence of characters that begins in the initial shift state. Each character shall be converted to a wide character as if by a call to
14302		

14303		the <i>mbrtowc()</i> function, with the conversion state described by an mbstate_t object
14304		initialized to zero before the first character is converted. The application shall ensure
14305		that the corresponding argument is a pointer to an array of wchar_t large enough to
14306		accept the sequence and the terminating null wide character, which shall be added
14307		automatically.
14308	[Matches a non-empty sequence of bytes from a set of expected bytes (the <i>scanset</i>). The
14309		normal skip over white-space characters shall be suppressed in this case. The
14310		application shall ensure that the corresponding argument is a pointer to the initial byte
14311		of an array of char , signed char , or unsigned char large enough to accept the sequence
14312		and a terminating null byte, which shall be added automatically.
14313		If an l (ell) qualifier is present, the input is a sequence of characters that begins in the
14314		initial shift state. Each character in the sequence shall be converted to a wide character
14315		as if by a call to the <i>mbrtowc()</i> function, with the conversion state described by an
14316		mbstate_t object initialized to zero before the first character is converted. The
14317		application shall ensure that the corresponding argument is a pointer to an array of
14318		wchar_t large enough to accept the sequence and the terminating null wide character,
14319		which shall be added automatically.
14320		The conversion specification includes all subsequent bytes in the <i>format</i> string up to
14321		and including the matching right square bracket (']'). The bytes between the square
14322		brackets (the <i>scanlist</i>) comprise the scanset, unless the byte after the left square bracket
14323		is a circumflex ('^'), in which case the scanset contains all bytes that do not appear in
14324		the scanlist between the circumflex and the right square bracket. If the conversion
14325		specification begins with "[^]" or "[^]", the right square bracket is included in the
14326		scanlist and the next right square bracket is the matching right square bracket that ends
14327		the conversion specification; otherwise, the first right square bracket is the one that
14328		ends the conversion specification. If a '-' is in the scanlist and is not the first character,
14329		nor the second where the first character is a '^', nor the last character, the behavior is
14330		implementation-defined.
14331	c	Matches a sequence of bytes of the number specified by the field width (1 if no field
14332		width is present in the conversion specification). The application shall ensure that the
14333		corresponding argument is a pointer to the initial byte of an array of char , signed char ,
14334		or unsigned char large enough to accept the sequence. No null byte is added. The
14335		normal skip over white-space characters shall be suppressed in this case.
14336		If an l (ell) qualifier is present, the input shall be a sequence of characters that begins in
14337		the initial shift state. Each character in the sequence is converted to a wide character
14338		as if by a call to the <i>mbrtowc()</i> function, with the conversion state described by an
14339		mbstate_t object initialized to zero before the first character is converted. The
14340		application shall ensure that the corresponding argument is a pointer to an array of
14341		wchar_t large enough to accept the resulting sequence of wide characters. No null wide
14342		character is added.
14343	p	Matches an implementation-defined set of sequences, which shall be the same as the set
14344		of sequences that is produced by the %p conversion specification of the corresponding
14345		<i>fprintf()</i> functions. The application shall ensure that the corresponding argument is a
14346		pointer to a pointer to void . The interpretation of the input item is implementation-
14347		defined. If the input item is a value converted earlier during the same program
14348		execution, the pointer that results shall compare equal to that value; otherwise, the
14349		behavior of the %p conversion specification is undefined.
14350	n	No input is consumed. The application shall ensure that the corresponding argument is
14351		a pointer to the integer into which shall be written the number of bytes read from the

14352 input so far by this call to the *fscanf()* functions. Execution of a *%n* conversion
 14353 specification shall not increment the assignment count returned at the completion of
 14354 execution of the function. No argument shall be converted, but one shall be consumed.
 14355 If the conversion specification includes an assignment-suppressing character or a field
 14356 width, the behavior is undefined.

14357 XSI C Equivalent to *lc*.

14358 XSI S Equivalent to *ls*.

14359 % Matches a single '*%*' character; no conversion or assignment occurs. The complete
 14360 conversion specification shall be *%%*.

14361 If a conversion specification is invalid, the behavior is undefined.

14362 The conversion specifiers A, E, F, G, and X are also valid and shall be equivalent to a, e, f, g, and
 14363 x, respectively.

14364 If end-of-file is encountered during input, conversion shall be terminated. If end-of-file occurs
 14365 before any bytes matching the current conversion specification (except for *%n*) have been read
 14366 (other than leading white-space characters, where permitted), execution of the current
 14367 conversion specification shall terminate with an input failure. Otherwise, unless execution of the
 14368 current conversion specification is terminated with a matching failure, execution of the
 14369 following conversion specification (if any) shall be terminated with an input failure.

14370 Reaching the end of the string in *sscanf()* shall be equivalent to encountering end-of-file for
 14371 *fscanf()*.

14372 If conversion terminates on a conflicting input, the offending input is left unread in the input.
 14373 Any trailing white space (including <newline>s) shall be left unread unless matched by a
 14374 conversion specification. The success of literal matches and suppressed assignments is only
 14375 directly determinable via the *%n* conversion specification.

14376 CX The *fscanf()* and *scanf()* functions may mark the *st_atime* field of the file associated with *stream*
 14377 for update. The *st_atime* field shall be marked for update by the first successful execution of
 14378 *fgetc()*, *fgets()*, *fread()*, *getc()*, *getchar()*, *gets()*, *fscanf()*, or *scanf()* using *stream* that returns data
 14379 not supplied by a prior call to *ungetc()*.

14380 RETURN VALUE

14381 Upon successful completion, these functions shall return the number of successfully matched
 14382 and assigned input items; this number can be zero in the event of an early matching failure. If
 14383 the input ends before the first matching failure or conversion, EOF shall be returned. If a read
 14384 CX error occurs, the error indicator for the stream is set, EOF shall be returned, and *errno* shall be set
 14385 to indicate the error.

14386 ERRORS

14387 For the conditions under which the *fscanf()* functions fail and may fail, refer to *fgetc()* or
 14388 *fgetwc()*.

14389 In addition, *fscanf()* may fail if:

14390 XSI [EILSEQ] Input byte sequence does not form a valid character.

14391 XSI [EINVAL] There are insufficient arguments.

14392 **EXAMPLES**

14393 The call:

```
14394       int i, n; float x; char name[50];
14395       n = scanf("%d%f%s", &i, &x, name);
```

14396 with the input line:

14397 25 54.32E-1 Hamster

14398 assigns to *n* the value 3, to *i* the value 25, to *x* the value 5.432, and *name* contains the string
 14399 "Hamster".

14400 The call:

```
14401       int i; float x; char name[50];
14402       (void) scanf("%2d%f*d %[0123456789]", &i, &x, name);
```

14403 with input:

14404 56789 0123 56a72

14405 assigns 56 to *i*, 789.0 to *x*, skips 0123, and places the string "56\0" in *name*. The next call to
 14406 *getchar()* shall return the character 'a'.

14407 **Reading Data into an Array**

14408 The following call uses *fscanf()* to read three floating-point numbers from standard input into
 14409 the *input* array.

```
14410       float input[3]; fscanf (stdin, "%f %f %f", input, input+1, input+2);
```

14411 **APPLICATION USAGE**

14412 If the application calling *fscanf()* has any objects of type **wint_t** or **wchar_t**, it must also include
 14413 the **<wchar.h>** header to have these objects defined.

14414 **RATIONALE**

14415 This function is aligned with the ISO/IEC 9899:1999 standard, and in doing so a few “obvious”
 14416 things were not included. Specifically, the set of characters allowed in a scanset is limited to
 14417 single-byte characters. In other similar places, multi-byte characters have been permitted, but
 14418 for alignment with the ISO/IEC 9899:1999 standard, it has not been done here. Applications
 14419 needing this could use the corresponding wide-character functions to achieve the desired
 14420 results.

14421 **FUTURE DIRECTIONS**

14422 None.

14423 **SEE ALSO**

14424 *getc()*, *printf()*, *setlocale()*, *strtod()*, *strtol()*, *strtoul()*, *wcrtomb()*, the Base Definitions volume of
 14425 IEEE Std 1003.1-2001, Chapter 7, Locale, **<langinfo.h>**, **<stdio.h>**, **<wchar.h>**

14426 **CHANGE HISTORY**

14427 First released in Issue 1. Derived from Issue 1 of the SVID.

14428 **Issue 5**

14429 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the **l** (ell) qualifier is
 14430 now defined for the **c**, **s**, and **[** conversion specifiers.

14431 The DESCRIPTION is updated to indicate that if infinity and NaN can be generated by the
 14432 *fprintf()* family of functions, then they are recognized by the *fscanf()* family.

14433 **Issue 6**

14434 The Open Group Corrigenda U021/7 and U028/10 are applied. These correct several
14435 occurrences of “characters” in the text which have been replaced with the term “bytes”.

14436 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

14437 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 14438 • The prototypes for *fscanf()*, *scanf()*, and *sscanf()* are updated.

- 14439 • The DESCRIPTION is updated.

- 14440 • The *hh*, *ll*, *j*, *t*, and *z* length modifiers are added.

- 14441 • The *a*, *A*, and *F* conversion characters are added.

14442 The DESCRIPTION is updated to use the terms “conversion specifier” and “conversion
14443 specification” consistently.

14444 NAME

14445 fseek, fseeko — reposition a file-position indicator in a stream

14446 SYNOPSIS

14447 #include <stdio.h>

14448 int fseek(FILE *stream, long offset, int whence);

14449 CX int fseeko(FILE *stream, off_t offset, int whence);

14450

14451 DESCRIPTION

14452 CX The functionality described on this reference page is aligned with the ISO C standard. Any
14453 conflict between the requirements described here and the ISO C standard is unintentional. This
14454 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

14455 The *fseek()* function shall set the file-position indicator for the stream pointed to by *stream*. If a
14456 read or write error occurs, the error indicator for the stream shall be set and *fseek()* fails.

14457 The new position, measured in bytes from the beginning of the file, shall be obtained by adding
14458 *offset* to the position specified by *whence*. The specified point is the beginning of the file for
14459 SEEK_SET, the current value of the file-position indicator for SEEK_CUR, or end-of-file for
14460 SEEK_END.

14461 If the stream is to be used with wide-character input/output functions, the application shall
14462 ensure that *offset* is either 0 or a value returned by an earlier call to *ftell()* on the same stream and
14463 *whence* is SEEK_SET.

14464 A successful call to *fseek()* shall clear the end-of-file indicator for the stream and undo any effects
14465 of *ungetc()* and *ungetwc()* on the same stream. After an *fseek()* call, the next operation on an
14466 update stream may be either input or output.

14467 CX If the most recent operation, other than *ftell()*, on a given stream is *fflush()*, the file offset in the
14468 underlying open file description shall be adjusted to reflect the location specified by *fseek()*.

14469 The *fseek()* function shall allow the file-position indicator to be set beyond the end of existing
14470 data in the file. If data is later written at this point, subsequent reads of data in the gap shall
14471 return bytes with the value 0 until data is actually written into the gap.

14472 The behavior of *fseek()* on devices which are incapable of seeking is implementation-defined.
14473 The value of the file offset associated with such a device is undefined.

14474 If the stream is writable and buffered data had not been written to the underlying file, *fseek()*
14475 shall cause the unwritten data to be written to the file and shall mark the *st_ctime* and *st_mtime*
14476 fields of the file for update.

14477 In a locale with state-dependent encoding, whether *fseek()* restores the stream's shift state is
14478 implementation-defined.

14479 The *fseeko()* function shall be equivalent to the *fseek()* function except that the *offset* argument is
14480 of type **off_t**.

14481 RETURN VALUE

14482 CX The *fseek()* and *fseeko()* functions shall return 0 if they succeed.

14483 CX Otherwise, they shall return -1 and set *errno* to indicate the error.

14484 ERRORS

14485 CX The *fseek()* and *fseeko()* functions shall fail if, either the *stream* is unbuffered or the *stream*'s
14486 buffer needed to be flushed, and the call to *fseek()* or *fseeko()* causes an underlying *lseek()* or
14487 *write()* to be invoked, and:

14488 CX	[EAGAIN]	The O_NONBLOCK flag is set for the file descriptor and the process would be delayed in the write operation.
14489		
14490 CX	[EBADF]	The file descriptor underlying the stream file is not open for writing or the stream's buffer needed to be flushed and the file is not open.
14491		
14492 CX	[EFBIG]	An attempt was made to write a file that exceeds the maximum file size.
14493 XSI	[EFBIG]	An attempt was made to write a file that exceeds the process' file size limit.
14494 CX	[EFBIG]	The file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.
14495		
14496 CX	[EINTR]	The write operation was terminated due to the receipt of a signal, and no data was transferred.
14497		
14498 CX	[EINVAL]	The <i>whence</i> argument is invalid. The resulting file-position indicator would be set to a negative value.
14499		
14500 CX	[EIO]	A physical I/O error has occurred, or the process is a member of a background process group attempting to perform a <i>write()</i> to its controlling terminal, TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions.
14501		
14502		
14503		
14504		
14505 CX	[ENOSPC]	There was no free space remaining on the device containing the file.
14506 CX	[ENXIO]	A request was made of a nonexistent device, or the request was outside the capabilities of the device.
14507		
14508 CX	[EOVERFLOW]	For <i>fseek()</i> , the resulting file offset would be a value which cannot be represented correctly in an object of type long .
14509		
14510 CX	[EOVERFLOW]	For <i>fseeko()</i> , the resulting file offset would be a value which cannot be represented correctly in an object of type off_t .
14511		
14512 CX	[EPIPE]	An attempt was made to write to a pipe or FIFO that is not open for reading by any process; a SIGPIPE signal shall also be sent to the thread.
14513		
14514 CX	[ESPIPE]	The file descriptor underlying <i>stream</i> is associated with a pipe or FIFO.

14515 **EXAMPLES**

14516 None.

14517 **APPLICATION USAGE**

14518 None.

14519 **RATIONALE**

14520 None.

14521 **FUTURE DIRECTIONS**

14522 None.

14523 **SEE ALSO**

14524 *fopen()*, *fsetpos()*, *ftell()*, *getrlimit()*, *lseek()*, *rewind()*, *ulimit()*, *ungetc()*, *write()*, the Base
 14525 Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

14526 **CHANGE HISTORY**

14527 First released in Issue 1. Derived from Issue 1 of the SVID.

14528 **Issue 5**

14529 Normative text previously in the APPLICATION USAGE section is moved to the
14530 DESCRIPTION.

14531 Large File Summit extensions are added.

14532 **Issue 6**

14533 Extensions beyond the ISO C standard are marked.

14534 The following new requirements on POSIX implementations derive from alignment with the
14535 Single UNIX Specification:

- 14536 • The *fseeko()* function is added.
- 14537 • The [EFBIG], [EOVERFLOW], and [ENXIO] mandatory error conditions are added.

14538 The following change is incorporated for alignment with the FIPS requirements:

- 14539 • The [EINTR] error is no longer an indication that the implementation does not report partial
14540 transfers.

14541 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

14542 The DESCRIPTION is updated to explicitly state that *fseek()* sets the file-position indicator, and
14543 then on error the error indicator is set and *fseek()* fails. This is for alignment with the
14544 ISO/IEC 9899:1999 standard.

14545 **NAME**

14546 fsetpos — set current file position

14547 **SYNOPSIS**

14548 #include <stdio.h>

14549 int fsetpos(FILE *stream, const fpos_t *pos);

14550 **DESCRIPTION**

14551 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 14552 conflict between the requirements described here and the ISO C standard is unintentional. This
 14553 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

14554 The *fsetpos()* function shall set the file position and state indicators for the stream pointed to by
 14555 *stream* according to the value of the object pointed to by *pos*, which the application shall ensure
 14556 is a value obtained from an earlier call to *fgetpos()* on the same stream. If a read or write error
 14557 occurs, the error indicator for the stream shall be set and *fsetpos()* fails.

14558 A successful call to the *fsetpos()* function shall clear the end-of-file indicator for the stream and
 14559 undo any effects of *ungetc()* on the same stream. After an *fsetpos()* call, the next operation on an
 14560 update stream may be either input or output.

14561 CX The behavior of *fsetpos()* on devices which are incapable of seeking is implementation-defined.
 14562 The value of the file offset associated with such a device is undefined.

14563 **RETURN VALUE**

14564 The *fsetpos()* function shall return 0 if it succeeds; otherwise, it shall return a non-zero value and
 14565 set *errno* to indicate the error.

14566 **ERRORS**

14567 CX The *fsetpos()* function shall fail if, either the *stream* is unbuffered or the *stream*'s buffer needed to
 14568 be flushed, and the call to *fsetpos()* causes an underlying *lseek()* or *write()* to be invoked, and:

14569 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor and the process would be
 14570 delayed in the write operation.

14571 CX [EBADF] The file descriptor underlying the stream file is not open for writing or the
 14572 stream's buffer needed to be flushed and the file is not open.

14573 CX [EFBIG] An attempt was made to write a file that exceeds the maximum file size.

14574 XSI [EFBIG] An attempt was made to write a file that exceeds the process' file size limit.

14575 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
 14576 offset maximum associated with the corresponding stream.

14577 CX [EINTR] The write operation was terminated due to the receipt of a signal, and no data
 14578 was transferred.

14579 CX [EINVAL] The *whence* argument is invalid. The resulting file-position indicator would be
 14580 set to a negative value.

14581 CX [EIO] A physical I/O error has occurred, or the process is a member of a
 14582 background process group attempting to perform a *write()* to its controlling
 14583 terminal, TOSTOP is set, the process is neither ignoring nor blocking
 14584 SIGTTOU, and the process group of the process is orphaned. This error may
 14585 also be returned under implementation-defined conditions.

14586 CX [ENOSPC] There was no free space remaining on the device containing the file.

14587 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
14588 capabilities of the device.

14589 CX [EPIPE] The file descriptor underlying *stream* is associated with a pipe or FIFO.

14590 CX [EPIPE] An attempt was made to write to a pipe or FIFO that is not open for reading
14591 by any process; a SIGPIPE signal shall also be sent to the thread.

14592 EXAMPLES

14593 None.

14594 APPLICATION USAGE

14595 None.

14596 RATIONALE

14597 None.

14598 FUTURE DIRECTIONS

14599 None.

14600 SEE ALSO

14601 *fopen()*, *ftell()*, *lseek()*, *rewind()*, *ungetc()*, *write()*, the Base Definitions volume of
14602 IEEE Std 1003.1-2001, <stdio.h>

14603 CHANGE HISTORY

14604 First released in Issue 4. Derived from the ISO C standard.

14605 Issue 6

14606 Extensions beyond the ISO C standard are marked.

14607 An additional [ESPIPE] error condition is added for sockets.

14608 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

14609 The DESCRIPTION is updated to clarify that the error indicator is set for the stream on a read or
14610 write error. This is for alignment with the ISO/IEC 9899:1999 standard.

14611 **NAME**

14612 fstat — get file status

14613 **SYNOPSIS**

14614 #include <sys/stat.h>

14615 int fstat(int *fildev*, struct stat **buf*);14616 **DESCRIPTION**14617 The *fstat()* function shall obtain information about an open file associated with the file
14618 descriptor *fildev*, and shall write it to the area pointed to by *buf*.14619 SHM If *fildev* references a shared memory object, the implementation shall update in the **stat** structure
14620 pointed to by the *buf* argument only the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields, and only the
14621 S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits need be
14622 valid. The implementation may update other fields and flags.14623 TYM If *fildev* references a typed memory object, the implementation shall update in the **stat** structure
14624 pointed to by the *buf* argument only the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields, and only the
14625 S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits need be
14626 valid. The implementation may update other fields and flags.14627 The *buf* argument is a pointer to a **stat** structure, as defined in <sys/stat.h>, into which
14628 information is placed concerning the file.14629 The structure members *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atime*, *st_ctime*, and *st_mtime*
14630 shall have meaningful values for all other file types defined in this volume of
14631 IEEE Std 1003.1-2001. The value of the member *st_nlink* shall be set to the number of links to the
14632 file.14633 An implementation that provides additional or alternative file access control mechanisms may,
14634 under implementation-defined conditions, cause *fstat()* to fail.14635 The *fstat()* function shall update any time-related fields as described in the Base Definitions
14636 volume of IEEE Std 1003.1-2001, Section 4.7, File Times Update, before writing into the **stat**
14637 structure.14638 **RETURN VALUE**14639 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
14640 indicate the error.14641 **ERRORS**14642 The *fstat()* function shall fail if:14643 [EBADF] The *fildev* argument is not a valid file descriptor.

14644 [EIO] An I/O error occurred while reading from the file system.

14645 [EOVERFLOW] The file size in bytes or the number of blocks allocated to the file or the file
14646 serial number cannot be represented correctly in the structure pointed to by
14647 *buf*.14648 The *fstat()* function may fail if:14649 [EOVERFLOW] One of the values is too large to store into the structure pointed to by the *buf*
14650 argument.

14651 **EXAMPLES**14652 **Obtaining File Status Information**

14653 The following example shows how to obtain file status information for a file named
 14654 `/home/cnd/mod1`. The structure variable *buffer* is defined for the **stat** structure. The
 14655 `/home/cnd/mod1` file is opened with read/write privileges and is passed to the open file
 14656 descriptor *fildev*.

```
14657 #include <sys/types.h>
14658 #include <sys/stat.h>
14659 #include <fcntl.h>

14660 struct stat buffer;
14661 int      status;
14662 ...
14663 fildev = open("/home/cnd/mod1", O_RDWR);
14664 status = fstat(fildev, &buffer);
```

14665 **APPLICATION USAGE**

14666 None.

14667 **RATIONALE**

14668 None.

14669 **FUTURE DIRECTIONS**

14670 None.

14671 **SEE ALSO**

14672 *lstat()*, *stat()*, the Base Definitions volume of IEEE Std 1003.1-2001, `<sys/stat.h>`, `<sys/types.h>`

14673 **CHANGE HISTORY**

14674 First released in Issue 1. Derived from Issue 1 of the SVID.

14675 **Issue 5**

14676 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

14677 Large File Summit extensions are added.

14678 **Issue 6**

14679 In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

14680 The following new requirements on POSIX implementations derive from alignment with the
 14681 Single UNIX Specification:

- 14682 • The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was
 14683 required for conforming implementations of previous POSIX specifications, it was not
 14684 required for UNIX applications.
- 14685 • The [EIO] mandatory error condition is added.
- 14686 • The [EOVERFLOW] mandatory error condition is added. This change is to support large
 14687 files.
- 14688 • The [EOVERFLOW] optional error condition is added.

14689 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that
 14690 shared memory object semantics apply to typed memory objects.

14691 **NAME**

14692 fstatvfs, statvfs — get file system information

14693 **SYNOPSIS**

14694 XSI #include <sys/statvfs.h>

14695 int fstatvfs(int *fildev*, struct statvfs **buf*);14696 int statvfs(const char *restrict *path*, struct statvfs *restrict *buf*);

14697

14698 **DESCRIPTION**14699 The *fstatvfs()* function shall obtain information about the file system containing the file
14700 referenced by *fildev*.14701 The *statvfs()* function shall obtain information about the file system containing the file named by
14702 *path*.14703 For both functions, the *buf* argument is a pointer to a **statvfs** structure that shall be filled. Read,
14704 write, or execute permission of the named file is not required.14705 The following flags can be returned in the *f_flag* member:

14706 ST_RDONLY Read-only file system.

14707 ST_NOSUID Setuid/setgid bits ignored by *exec*.14708 It is unspecified whether all members of the **statvfs** structure have meaningful values on all file
14709 systems.14710 **RETURN VALUE**14711 Upon successful completion, *statvfs()* shall return 0. Otherwise, it shall return -1 and set *errno* to
14712 indicate the error.14713 **ERRORS**14714 The *fstatvfs()* and *statvfs()* functions shall fail if:

14715 [EIO] An I/O error occurred while reading the file system.

14716 [EINTR] A signal was caught during execution of the function.

14717 [EOVERFLOW] One of the values to be returned cannot be represented correctly in the
14718 structure pointed to by *buf*.14719 The *fstatvfs()* function shall fail if:14720 [EBADF] The *fildev* argument is not an open file descriptor.14721 The *statvfs()* function shall fail if:

14722 [EACCES] Search permission is denied on a component of the path prefix.

14723 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
14724 argument.

14725 [ENAMETOOLONG] The length of a pathname exceeds {PATH_MAX} or a pathname component is

14726 longer than {NAME_MAX}.
1472714728 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.14729 [ENOTDIR] A component of the path prefix of *path* is not a directory.14730 The *statvfs()* function may fail if:

14731 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 14732 resolution of the *path* argument.

14733 [ENAMETOOLONG]
 14734 Pathname resolution of a symbolic link produced an intermediate result
 14735 whose length exceeds {PATH_MAX}.

14736 EXAMPLES

14737 Obtaining File System Information Using *fstatvfs()*

14738 The following example shows how to obtain file system information for the file system upon
 14739 which the file named */home/cnd/mod1* resides, using the *fstatvfs()* function. The
 14740 */home/cnd/mod1* file is opened with read/write privileges and the open file descriptor is passed
 14741 to the *fstatvfs()* function.

```
14742 #include <statvfs.h>
14743 #include <fcntl.h>

14744 struct statvfs buffer;
14745 int          status;
14746 ...
14747 fildes = open("/home/cnd/mod1", O_RDWR);
14748 status  = fstatvfs(fildes, &buffer);
```

14749 Obtaining File System Information Using *statvfs()*

14750 The following example shows how to obtain file system information for the file system upon
 14751 which the file named */home/cnd/mod1* resides, using the *statvfs()* function.

```
14752 #include <statvfs.h>

14753 struct statvfs buffer;
14754 int          status;
14755 ...
14756 status = statvfs("/home/cnd/mod1", &buffer);
```

14757 APPLICATION USAGE

14758 None.

14759 RATIONALE

14760 None.

14761 FUTURE DIRECTIONS

14762 None.

14763 SEE ALSO

14764 *chmod()*, *chown()*, *creat()*, *dup()*, *exec*, *fcntl()*, *link()*, *mknod()*, *open()*, *pipe()*, *read()*, *time()*,
 14765 *unlink()*, *utime()*, *write()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**sys/statvfs.h**>

14766 CHANGE HISTORY

14767 First released in Issue 4, Version 2.

14768 Issue 5

14769 Moved from X/OPEN UNIX extension to BASE.

14770 Large File Summit extensions are added.

14771 **Issue 6**

14772 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

14773 The **restrict** keyword is added to the *statvfs()* prototype for alignment with the
14774 ISO/IEC 9899:1999 standard.

14775 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
14776 [ELOOP] error condition is added.

14777 **NAME**

14778 fsync — synchronize changes to a file

14779 **SYNOPSIS**

14780 FSC #include <unistd.h>

14781 int fsync(int *fil*des);

14782

14783 **DESCRIPTION**

14784 The *fsync()* function shall request that all data for the open file descriptor named by *fil*des is to be
 14785 transferred to the storage device associated with the file described by *fil*des in an
 14786 implementation-defined manner. The *fsync()* function shall not return until the system has
 14787 completed that action or until an error is detected.

14788 SIO If _POSIX_SYNCHRONIZED_IO is defined, the *fsync()* function shall force all currently queued
 14789 I/O operations associated with the file indicated by file descriptor *fil*des to the synchronized I/O
 14790 completion state. All I/O operations shall be completed as defined for synchronized I/O file
 14791 integrity completion.

14792 **RETURN VALUE**

14793 Upon successful completion, *fsync()* shall return 0. Otherwise, -1 shall be returned and *errno* set
 14794 to indicate the error. If the *fsync()* function fails, outstanding I/O operations are not guaranteed
 14795 to have been completed.

14796 **ERRORS**14797 The *fsync()* function shall fail if:14798 [EBADF] The *fil*des argument is not a valid descriptor.14799 [EINTR] The *fsync()* function was interrupted by a signal.14800 [EINVAL] The *fil*des argument does not refer to a file on which this operation is possible.

14801 [EIO] An I/O error occurred while reading from or writing to the file system.

14802 In the event that any of the queued I/O operations fail, *fsync()* shall return the error conditions
 14803 defined for *read()* and *write()*.

14804 **EXAMPLES**

14805 None.

14806 **APPLICATION USAGE**

14807 The *fsync()* function should be used by programs which require modifications to a file to be
 14808 completed before continuing; for example, a program which contains a simple transaction
 14809 facility might use it to ensure that all modifications to a file or files caused by a transaction are
 14810 recorded.

14811 **RATIONALE**

14812 The *fsync()* function is intended to force a physical write of data from the buffer cache, and to
 14813 assure that after a system crash or other failure that all data up to the time of the *fsync()* call is
 14814 recorded on the disk. Since the concepts of “buffer cache”, “system crash”, “physical write”, and
 14815 “non-volatile storage” are not defined here, the wording has to be more abstract.

14816 If _POSIX_SYNCHRONIZED_IO is not defined, the wording relies heavily on the conformance
 14817 document to tell the user what can be expected from the system. It is explicitly intended that a
 14818 null implementation is permitted. This could be valid in the case where the system cannot assure
 14819 non-volatile storage under any circumstances or when the system is highly fault-tolerant and the
 14820 functionality is not required. In the middle ground between these extremes, *fsync()* might or
 14821 might not actually cause data to be written where it is safe from a power failure. The

14822 conformance document should identify at least that one configuration exists (and how to obtain
14823 that configuration) where this can be assured for at least some files that the user can select to use
14824 for critical data. It is not intended that an exhaustive list is required, but rather sufficient
14825 information is provided so that if critical data needs to be saved, the user can determine how the
14826 system is to be configured to allow the data to be written to non-volatile storage.

14827 It is reasonable to assert that the key aspects of *fsync()* are unreasonable to test in a test suite.
14828 That does not make the function any less valuable, just more difficult to test. A formal
14829 conformance test should probably force a system crash (power shutdown) during the test for
14830 this condition, but it needs to be done in such a way that automated testing does not require this
14831 to be done except when a formal record of the results is being made. It would also not be
14832 unreasonable to omit testing for *fsync()*, allowing it to be treated as a quality-of-implementation
14833 issue.

14834 **FUTURE DIRECTIONS**

14835 None.

14836 **SEE ALSO**

14837 *sync()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

14838 **CHANGE HISTORY**

14839 First released in Issue 3.

14840 **Issue 5**

14841 Aligned with *fsync()* in the POSIX Realtime Extension. Specifically, the DESCRIPTION and
14842 RETURN VALUE sections are much expanded, and the ERRORS section is updated to indicate
14843 that *fsync()* can return the error conditions defined for *read()* and *write()*.

14844 **Issue 6**

14845 This function is marked as part of the File Synchronization option.

14846 The following new requirements on POSIX implementations derive from alignment with the
14847 Single UNIX Specification:

- 14848 • The [EINVAL] and [EIO] mandatory error conditions are added.

14849 **NAME**

14850 ftell, ftello — return a file offset in a stream

14851 **SYNOPSIS**

14852 #include <stdio.h>

14853 long ftell(FILE *stream);

14854 CX off_t ftello(FILE *stream);

14855

14856 **DESCRIPTION**

14857 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 14858 conflict between the requirements described here and the ISO C standard is unintentional. This
 14859 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

14860 The *ftell()* function shall obtain the current value of the file-position indicator for the stream
 14861 pointed to by *stream*.

14862 CX The *ftello()* function shall be equivalent to *ftell()*, except that the return value is of type **off_t**.

14863 **RETURN VALUE**

14864 CX Upon successful completion, *ftell()* and *ftello()* shall return the current value of the file-position
 14865 indicator for the stream measured in bytes from the beginning of the file.

14866 CX Otherwise, *ftell()* and *ftello()* shall return -1 , cast to **long** and **off_t** respectively, and set *errno* to
 14867 indicate the error.

14868 **ERRORS**

14869 CX The *ftell()* and *ftello()* functions shall fail if:

14870 CX [EBADF] The file descriptor underlying *stream* is not an open file descriptor.

14871 CX [EOVERFLOW] For *ftell()*, the current file offset cannot be represented correctly in an object of
 14872 type **long**.

14873 CX [EOVERFLOW] For *ftello()*, the current file offset cannot be represented correctly in an object
 14874 of type **off_t**.

14875 CX [ESPIPE] The file descriptor underlying *stream* is associated with a pipe or FIFO.

14876 The *ftell()* function may fail if:

14877 CX [ESPIPE] The file descriptor underlying *stream* is associated with a socket.

14878 **EXAMPLES**

14879 None.

14880 **APPLICATION USAGE**

14881 None.

14882 **RATIONALE**

14883 None.

14884 **FUTURE DIRECTIONS**

14885 None.

14886 **SEE ALSO**

14887 *fgetpos()*, *fopen()*, *fseek()*, *lseek()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

14888 **CHANGE HISTORY**

14889 First released in Issue 1. Derived from Issue 1 of the SVID.

14890 **Issue 5**

14891 Large File Summit extensions are added.

14892 **Issue 6**

14893 Extensions beyond the ISO C standard are marked.

14894 The following new requirements on POSIX implementations derive from alignment with the
14895 Single UNIX Specification:

- 14896 • The *ftello()* function is added.
- 14897 • The [EOVERFLOW] error conditions are added.

14898 An additional [ESPIPE] error condition is added for sockets.

14899 NAME

14900 ftime — get date and time (**LEGACY**)

14901 SYNOPSIS

14902 xSI `#include <sys/timeb.h>`

14903 `int ftime(struct timeb *tp);`

14904

14905 DESCRIPTION

14906 The *ftime()* function shall set the *time* and *millitm* members of the **timeb** structure pointed to by
 14907 *tp* to contain the seconds and milliseconds portions, respectively, of the current time in seconds
 14908 since the Epoch. The contents of the *timezone* and *dstflag* members of *tp* after a call to *ftime()* are
 14909 unspecified.

14910 The system clock need not have millisecond granularity. Depending on any granularity
 14911 (particularly a granularity of one) renders code non-portable.

14912 RETURN VALUE

14913 Upon successful completion, the *ftime()* function shall return 0; otherwise, -1 shall be returned.

14914 ERRORS

14915 No errors are defined.

14916 EXAMPLES

14917 Getting the Current Time and Date

14918 The following example shows how to get the current system time values using the *ftime()*
 14919 function. The **timeb** structure pointed to by *tp* is filled with the current system time values for
 14920 *time* and *millitm*.

14921 `#include <sys/timeb.h>`

14922 `struct timeb tp;`

14923 `int status;`

14924 `...`

14925 `status = ftime(&tp);`

14926 APPLICATION USAGE

14927 For applications portability, the *time()* function should be used to determine the current time
 14928 instead of *ftime()*. Realtime applications should use *clock_gettime()* to determine the current
 14929 time instead of *ftime()*.

14930 RATIONALE

14931 None.

14932 FUTURE DIRECTIONS

14933 This function may be withdrawn in a future version.

14934 SEE ALSO

14935 *clock_getres()*, *ctime()*, *gettimeofday()*, *time()*, the Base Definitions volume of
 14936 IEEE Std 1003.1-2001, <sys/timeb.h>

14937 CHANGE HISTORY

14938 First released in Issue 4, Version 2.

14939 **Issue 5**

14940 Moved from X/OPEN UNIX extension to BASE.

14941 Normative text previously in the APPLICATION USAGE section is moved to the
14942 DESCRIPTION.

14943 **Issue 6**

14944 This function is marked LEGACY.

14945 The DESCRIPTION is updated to refer to “seconds since the Epoch” rather than “seconds since
14946 00:00:00 UTC (Coordinated Universal Time), January 1 1970” for consistency with other *time*
14947 functions.

14948 NAME

14949 ftok — generate an IPC key

14950 SYNOPSIS

14951 xSI #include <sys/ipc.h>

14952 key_t ftok(const char *path, int id);

14953

14954 DESCRIPTION

14955 The *ftok()* function shall return a key based on *path* and *id* that is usable in subsequent calls to
 14956 *msgget()*, *semget()*, and *shmget()*. The application shall ensure that the *path* argument is the
 14957 pathname of an existing file that the process is able to *stat()*.

14958 The *ftok()* function shall return the same key value for all paths that name the same file, when
 14959 called with the same *id* value, and return different key values when called with different *id*
 14960 values or with paths that name different files existing on the same file system at the same time. It
 14961 is unspecified whether *ftok()* shall return the same key value when called again after the file
 14962 named by *path* is removed and recreated with the same name.

14963 Only the low-order 8-bits of *id* are significant. The behavior of *ftok()* is unspecified if these bits
 14964 are 0.

14965 RETURN VALUE

14966 Upon successful completion, *ftok()* shall return a key. Otherwise, *ftok()* shall return (**key_t**)−1
 14967 and set *errno* to indicate the error.

14968 ERRORS

14969 The *ftok()* function shall fail if:

14970 [EACCES] Search permission is denied for a component of the path prefix.

14971 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 14972 argument.

14973 [ENAMETOOLONG]

14974 The length of the *path* argument exceeds {PATH_MAX} or a pathname
 14975 component is longer than {NAME_MAX}.

14976 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

14977 [ENOTDIR] A component of the path prefix is not a directory.

14978 The *ftok()* function may fail if:

14979 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 14980 resolution of the *path* argument.

14981 [ENAMETOOLONG]

14982 Pathname resolution of a symbolic link produced an intermediate result
 14983 whose length exceeds {PATH_MAX}.

14984 **EXAMPLES**14985 **Getting an IPC Key**

14986 The following example gets a unique key that can be used by the IPC functions *semget()*,
 14987 *msgget()*, and *shmget()*. The key returned by *ftok()* for this example is based on the ID value *S*
 14988 and the pathname */tmp*.

```
14989 #include <sys/ipc.h>
14990 ...
14991 key_t key;
14992 char *path = "/tmp";
14993 int id = 'S';
14994 key = ftok(path, id);
```

14995 **Saving an IPC Key**

14996 The following example gets a unique key based on the pathname */tmp* and the ID value *a*. It
 14997 also assigns the value of the resulting key to the *semkey* variable so that it will be available to a
 14998 later call to *semget()*, *msgget()*, or *shmget()*.

```
14999 #include <sys/ipc.h>
15000 ...
15001 key_t semkey;
15002 if ((semkey = ftok("/tmp", 'a')) == (key_t) -1) {
15003     perror("IPC error: ftok"); exit(1);
15004 }
```

15005 **APPLICATION USAGE**

15006 For maximum portability, *id* should be a single-byte character.

15007 **RATIONALE**

15008 None.

15009 **FUTURE DIRECTIONS**

15010 None.

15011 **SEE ALSO**

15012 *msgget()*, *semget()*, *shmget()*, the Base Definitions volume of IEEE Std 1003.1-2001, *<sys/ipc.h>*

15013 **CHANGE HISTORY**

15014 First released in Issue 4, Version 2.

15015 **Issue 5**

15016 Moved from X/OPEN UNIX extension to BASE.

15017 **Issue 6**

15018 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

15019 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
 15020 [ELOOP] error condition is added.

15021 **NAME**

15022 ftruncate — truncate a file to a specified length

15023 **SYNOPSIS**

15024 #include <unistd.h>

15025 int ftruncate(int *fildes*, off_t *length*);15026 **DESCRIPTION**15027 If *fildes* is not a valid file descriptor open for writing, the *ftruncate()* function shall fail.

15028 If *fildes* refers to a regular file, the *ftruncate()* function shall cause the size of the file to be truncated to *length*. If the size of the file previously exceeded *length*, the extra data shall no longer be available to reads on the file. If the file previously was smaller than this size, *ftruncate()* shall either increase the size of the file or fail. XSI-conformant systems shall increase the size of the file. If the file size is increased, the extended area shall appear as if it were zero-filled. The value of the seek pointer shall not be modified by a call to *ftruncate()*.

15034 Upon successful completion, if *fildes* refers to a regular file, the *ftruncate()* function shall mark for update the *st_ctime* and *st_mtime* fields of the file and the S_ISUID and S_ISGID bits of the file mode may be cleared. If the *ftruncate()* function is unsuccessful, the file is unaffected.

15037 XSI If the request would cause the file size to exceed the soft file size limit for the process, the request shall fail and the implementation shall generate the SIGXFSZ signal for the thread.

15039 If *fildes* refers to a directory, *ftruncate()* shall fail.15040 If *fildes* refers to any other file type, except a shared memory object, the result is unspecified.

15041 SHM If *fildes* refers to a shared memory object, *ftruncate()* shall set the size of the shared memory object to *length*.

15043 MF|SHM If the effect of *ftruncate()* is to decrease the size of a shared memory object or memory mapped file and whole pages beyond the new end were previously mapped, then the whole pages beyond the new end shall be discarded.

15046 MPR If the Memory Protection option is supported, references to discarded pages shall result in the generation of a SIGBUS signal; otherwise, the result of such references is undefined.

15048 MF|SHM If the effect of *ftruncate()* is to increase the size of a shared memory object, it is unspecified whether the contents of any mapped pages between the old end-of-file and the new are flushed to the underlying object.

15051 **RETURN VALUE**

15052 Upon successful completion, *ftruncate()* shall return 0; otherwise, -1 shall be returned and *errno* set to indicate the error.

15054 **ERRORS**15055 The *ftruncate()* function shall fail if:

15056 [EINTR] A signal was caught during execution.

15057 [EINVAL] The *length* argument was less than 0.

15058 [EFBIG] or [EINVAL]

15059 The *length* argument was greater than the maximum file size.

15060 XSI [EFBIG] The file is a regular file and *length* is greater than the offset maximum established in the open file description associated with *fildes*.

15062 [EIO] An I/O error occurred while reading from or writing to a file system.

15063 [EBADF] or [EINVAL]
 15064 The *fildest* argument is not a file descriptor open for writing.
 15065 [EINVAL] The *fildest* argument references a file that was opened without write
 15066 permission.
 15067 [EROFS] The named file resides on a read-only file system.

15068 **EXAMPLES**

15069 None.

15070 **APPLICATION USAGE**

15071 None.

15072 **RATIONALE**

15073 The *ftruncate()* function is part of IEEE Std 1003.1-2001 as it was deemed to be more useful than
 15074 *truncate()*. The *truncate()* function is provided as an XSI extension.

15075 **FUTURE DIRECTIONS**

15076 None.

15077 **SEE ALSO**15078 *open()*, *truncate()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>15079 **CHANGE HISTORY**

15080 First released in Issue 4, Version 2.

15081 **Issue 5**

15082 Moved from X/OPEN UNIX extension to BASE and aligned with *ftruncate()* in the POSIX
 15083 Realtime Extension. Specifically, the DESCRIPTION is extensively reworded and [EROFS] is
 15084 added to the list of mandatory errors that can be returned by *ftruncate()*.

15085 Large File Summit extensions are added.

15086 **Issue 6**15087 The *truncate()* function has been split out into a separate reference page.

15088 The following new requirements on POSIX implementations derive from alignment with the
 15089 Single UNIX Specification:

- 15090 • The DESCRIPTION is changed to indicate that if the file size is changed, and if the file is a
 15091 regular file, the S_ISUID and S_ISGID bits in the file mode may be cleared.

15092 The following changes were made to align with the IEEE P1003.1a draft standard:

- 15093 • The DESCRIPTION text is updated.

15094 XSI-conformant systems are required to increase the size of the file if the file was previously
 15095 smaller than the size requested.

15096 NAME

15097 ftrylockfile — stdio locking functions

15098 SYNOPSIS

15099 TSF #include <stdio.h>

15100 int ftrylockfile(FILE *file);

15101

15102 DESCRIPTION

15103 Refer to *flockfile()*.

15104 **NAME**

15105 ftw — traverse (walk) a file tree

15106 **SYNOPSIS**

15107 XSI #include <ftw.h>

```
15108       int ftw(const char *path, int (*fn)(const char *,
15109       const struct stat *ptr, int flag), int ndirs);
```

15110

15111 **DESCRIPTION**

15112 The *ftw()* function shall recursively descend the directory hierarchy rooted in *path*. For each
 15113 object in the hierarchy, *ftw()* shall call the function pointed to by *fn*, passing it a pointer to a
 15114 null-terminated character string containing the name of the object, a pointer to a **stat** structure
 15115 containing information about the object, and an integer. Possible values of the integer, defined
 15116 in the <ftw.h> header, are:

15117 FTW_D For a directory.

15118 FTW_DNR For a directory that cannot be read.

15119 FTW_F For a file.

15120 FTW_SL For a symbolic link (but see also FTW_NS below).

15121 FTW_NS For an object other than a symbolic link on which *stat()* could not successfully be
 15122 executed. If the object is a symbolic link and *stat()* failed, it is unspecified whether
 15123 *ftw()* passes FTW_SL or FTW_NS to the user-supplied function.

15124 If the integer is FTW_DNR, descendants of that directory shall not be processed. If the integer is
 15125 FTW_NS, the **stat** structure contains undefined values. An example of an object that would
 15126 cause FTW_NS to be passed to the function pointed to by *fn* would be a file in a directory with
 15127 read but without execute (search) permission.

15128 The *ftw()* function shall visit a directory before visiting any of its descendants.15129 The *ftw()* function shall use at most one file descriptor for each level in the tree.15130 The argument *ndirs* should be in the range [1,{OPEN_MAX}].

15131 The tree traversal shall continue until either the tree is exhausted, an invocation of *fn* returns a
 15132 non-zero value, or some error, other than [EACCES], is detected within *ftw()*.

15133 The *ndirs* argument shall specify the maximum number of directory streams or file descriptors
 15134 or both available for use by *ftw()* while traversing the tree. When *ftw()* returns it shall close any
 15135 directory streams and file descriptors it uses not counting any opened by the application-
 15136 supplied *fn* function.

15137 The results are unspecified if the application-supplied *fn* function does not preserve the current
 15138 working directory.

15139 The *ftw()* function need not be reentrant. A function that is not required to be reentrant is not
 15140 required to be thread-safe.

15141 **RETURN VALUE**

15142 If the tree is exhausted, *ftw()* shall return 0. If the function pointed to by *fn* returns a non-zero
 15143 value, *ftw()* shall stop its tree traversal and return whatever value was returned by the function
 15144 pointed to by *fn*. If *ftw()* detects an error, it shall return -1 and set *errno* to indicate the error.

15145 If *ftw()* encounters an error other than [EACCES] (see FTW_DNR and FTW_NS above), it shall
 15146 return -1 and set *errno* to indicate the error. The external variable *errno* may contain any error

15147 value that is possible when a directory is opened or when one of the *stat* functions is executed on
 15148 a directory or file.

15149 ERRORS

15150 The *ftw()* function shall fail if:

15151 [EACCES] Search permission is denied for any component of *path* or read permission is
 15152 denied for *path*.

15153 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 15154 argument.

15155 [ENAMETOOLONG]
 15156 The length of the *path* argument exceeds {PATH_MAX} or a pathname
 15157 component is longer than {NAME_MAX}.

15158 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

15159 [ENOTDIR] A component of *path* is not a directory.

15160 [EOVERFLOW] A field in the **stat** structure cannot be represented correctly in the current
 15161 programming environment for one or more files found in the file hierarchy.

15162 The *ftw()* function may fail if:

15163 [EINVAL] The value of the *ndirs* argument is invalid.

15164 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 15165 resolution of the *path* argument.

15166 [ENAMETOOLONG]
 15167 Pathname resolution of a symbolic link produced an intermediate result
 15168 whose length exceeds {PATH_MAX}.

15169 In addition, if the function pointed to by *fn* encounters system errors, *errno* may be set
 15170 accordingly.

15171 EXAMPLES

15172 Walking a Directory Structure

15173 The following example walks the current directory structure, calling the *fn* function for every
 15174 directory entry, using at most 10 file descriptors:

```
15175 #include <ftw.h>
15176 ...
15177 if (ftw(".", fn, 10) != 0) {
15178     perror("ftw"); exit(2);
15179 }
```

15180 APPLICATION USAGE

15181 The *ftw()* function may allocate dynamic storage during its operation. If *ftw()* is forcibly
 15182 terminated, such as by *longjmp()* or *siglongjmp()* being executed by the function pointed to by *fn*
 15183 or an interrupt routine, *ftw()* does not have a chance to free that storage, so it remains
 15184 permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has
 15185 occurred, and arrange to have the function pointed to by *fn* return a non-zero value at its next
 15186 invocation.

15187 **RATIONALE**

15188 None.

15189 **FUTURE DIRECTIONS**

15190 None.

15191 **SEE ALSO**

15192 *longjmp()*, *lstat()*, *malloc()*, *nftw()*, *opendir()*, *siglongjmp()*, *stat()*, the Base Definitions volume of
15193 IEEE Std 1003.1-2001, <ftw.h>, <sys/stat.h>

15194 **CHANGE HISTORY**

15195 First released in Issue 1. Derived from Issue 1 of the SVID.

15196 **Issue 5**

15197 UX codings in the DESCRIPTION, RETURN VALUE, and ERRORS sections are changed to EX.

15198 **Issue 6**

15199 The ERRORS section is updated as follows:

- 15200 • The wording of the mandatory [ELOOP] error condition is updated.
- 15201 • A second optional [ELOOP] error condition is added.
- 15202 • The [EOVERFLOW] mandatory error condition is added.

15203 Text is added to the DESCRIPTION to say that the *ftw()* function need not be reentrant and that
15204 the results are unspecified if the application-supplied *fn* function does not preserve the current
15205 working directory.

15206 NAME

15207 funlockfile — stdio locking functions

15208 SYNOPSIS

15209 TSF `#include <stdio.h>`

15210 `void funlockfile(FILE *file);`

15211

15212 DESCRIPTION

15213 Refer to *flockfile()*.

15214 **NAME**

15215 fwide — set stream orientation

15216 **SYNOPSIS**

15217 #include <stdio.h>

15218 #include <wchar.h>

15219 int fwide(FILE **stream*, int *mode*);15220 **DESCRIPTION**

15221 CX The functionality described on this reference page is aligned with the ISO C standard. Any
15222 conflict between the requirements described here and the ISO C standard is unintentional. This
15223 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

15224 The *fwide()* function shall determine the orientation of the stream pointed to by *stream*. If *mode* is
15225 greater than zero, the function first attempts to make the stream wide-oriented. If *mode* is less
15226 than zero, the function first attempts to make the stream byte-oriented. Otherwise, *mode* is zero
15227 and the function does not alter the orientation of the stream.

15228 If the orientation of the stream has already been determined, *fwide()* shall not change it.

15229 CX Since no return value is reserved to indicate an error, an application wishing to check for error
15230 situations should set *errno* to 0, then call *fwide()*, then check *errno*, and if it is non-zero, assume
15231 an error has occurred.

15232 **RETURN VALUE**

15233 The *fwide()* function shall return a value greater than zero if, after the call, the stream has wide-
15234 orientation, a value less than zero if the stream has byte-orientation, or zero if the stream has no
15235 orientation.

15236 **ERRORS**

15237 The *fwide()* function may fail if:

15238 CX [EBADF] The *stream* argument is not a valid stream.

15239 **EXAMPLES**

15240 None.

15241 **APPLICATION USAGE**

15242 A call to *fwide()* with *mode* set to zero can be used to determine the current orientation of a
15243 stream.

15244 **RATIONALE**

15245 None.

15246 **FUTURE DIRECTIONS**

15247 None.

15248 **SEE ALSO**

15249 The Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

15250 **CHANGE HISTORY**

15251 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
15252 (E).

15253 **Issue 6**

15254 Extensions beyond the ISO C standard are marked.

15255 NAME

15256 fwprintf, swprintf, wprintf — print formatted wide-character output

15257 SYNOPSIS

15258 #include <stdio.h>

15259 #include <wchar.h>

15260 int fwprintf(FILE *restrict stream, const wchar_t *restrict format, ...);

15261 int swprintf(wchar_t *restrict ws, size_t n,

15262 const wchar_t *restrict format, ...);

15263 int wprintf(const wchar_t *restrict format, ...);

15264 DESCRIPTION

15265 CX The functionality described on this reference page is aligned with the ISO C standard. Any
15266 conflict between the requirements described here and the ISO C standard is unintentional. This
15267 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

15268 The *fwprintf()* function shall place output on the named output *stream*. The *wprintf()* function
15269 shall place output on the standard output stream *stdout*. The *swprintf()* function shall place
15270 output followed by the null wide character in consecutive wide characters starting at **ws*; no
15271 more than *n* wide characters shall be written, including a terminating null wide character, which
15272 is always added (unless *n* is zero).

15273 Each of these functions shall convert, format, and print its arguments under control of the *format*
15274 wide-character string. The *format* is composed of zero or more directives: *ordinary wide-*
15275 *characters*, which are simply copied to the output stream, and *conversion specifications*, each of
15276 which results in the fetching of zero or more arguments. The results are undefined if there are
15277 insufficient arguments for the *format*. If the *format* is exhausted while arguments remain, the
15278 excess arguments are evaluated but are otherwise ignored.

15279 XSI Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than
15280 to the next unused argument. In this case, the conversion specifier wide character % (see below)
15281 is replaced by the sequence "%n\$", where *n* is a decimal integer in the range [1,{NL_ARGMAX}],
15282 giving the position of the argument in the argument list. This feature provides for the definition
15283 of *format* wide-character strings that select arguments in an order appropriate to specific
15284 languages (see the EXAMPLES section).

15285 The *format* can contain either numbered argument specifications (that is, "%n\$" and "%m\$"), or
15286 unnumbered argument conversion specifications (that is, % and *), but not both. The only
15287 exception to this is that %% can be mixed with the "%n\$" form. The results of mixing numbered
15288 and unnumbered argument specifications in a *format* wide-character string are undefined. When
15289 numbered argument specifications are used, specifying the *N*th argument requires that all the
15290 leading arguments, from the first to the (*N*−1)th, are specified in the *format* wide-character string.

15291 In *format* wide-character strings containing the "%n\$" form of conversion specification,
15292 numbered arguments in the argument list can be referenced from the *format* wide-character
15293 string as many times as required.

15294 In *format* wide-character strings containing the % form of conversion specification, each
15295 argument in the argument list shall be used exactly once.

15296 CX All forms of the *fwprintf()* function allow for the insertion of a locale-dependent radix character
15297 in the output string, output as a wide-character value. The radix character is defined in the
15298 program's locale (category *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix
15299 character is not defined, the radix character shall default to a period ('.').

15300 XSI Each conversion specification is introduced by the '%' wide character or by the wide-character
15301 sequence "%n\$", after which the following appear in sequence:

- 15302 • Zero or more *flags* (in any order), which modify the meaning of the conversion specification.
- 15303 • An optional minimum *field width*. If the converted value has fewer wide characters than the
- 15304 field width, it shall be padded with spaces by default on the left; it shall be padded on the
- 15305 right, if the left-adjustment flag ('-'), described below, is given to the field width. The field
- 15306 width takes the form of an asterisk ('*'), described below, or a decimal integer.
- 15307 • An optional *precision* that gives the minimum number of digits to appear for the d, i, o, u, x,
- 15308 and X conversion specifiers; the number of digits to appear after the radix character for the a,
- 15309 A, e, E, f, and F conversion specifiers; the maximum number of significant digits for the g
- 15310 and G conversion specifiers; or the maximum number of wide characters to be printed from a
- 15311 string in the s conversion specifiers. The precision takes the form of a period ('.') followed
- 15312 either by an asterisk ('*'), described below, or an optional decimal digit string, where a null
- 15313 digit string is treated as 0. If a precision appears with any other conversion wide character,
- 15314 the behavior is undefined.
- 15315 • An optional length modifier that specifies the size of the argument.
- 15316 • A *conversion specifier* wide character that indicates the type of conversion to be applied.
- 15317 A field width, or precision, or both, may be indicated by an asterisk ('*'). In this case an
- 15318 argument of type **int** supplies the field width or precision. Applications shall ensure that
- 15319 arguments specifying field width, or precision, or both appear in that order before the argument,
- 15320 if any, to be converted. A negative field width is taken as a '-' flag followed by a positive field
- 15321 XSI width. A negative precision is taken as if the precision were omitted. In *format* wide-character
- 15322 strings containing the "%n\$" form of a conversion specification, a field width or precision may
- 15323 be indicated by the sequence "*m\$", where *m* is a decimal integer in the range
- 15324 [1,{NL_ARGMAX}] giving the position in the argument list (after the *format* argument) of an
- 15325 integer argument containing the field width or precision, for example:
- 15326

```
wprintf(L"%1$d:%2$.3$d:%4$.3$d\n", hour, min, precision, sec);
```
- 15327 The flag wide characters and their meanings are:
- 15328 XSI ' The integer portion of the result of a decimal conversion (%i, %d, %u, %f, %F, %g, or %G)
- 15329 shall be formatted with thousands' grouping wide characters. For other conversions,
- 15330 the behavior is undefined. The numeric grouping wide character is used.
- 15331 - The result of the conversion shall be left-justified within the field. The conversion shall
- 15332 be right-justified if this flag is not specified.
- 15333 + The result of a signed conversion shall always begin with a sign ('+' or '-'). The
- 15334 conversion shall begin with a sign only when a negative value is converted if this flag is
- 15335 not specified.
- 15336 <space> If the first wide character of a signed conversion is not a sign, or if a signed conversion
- 15337 results in no wide characters, a <space> shall be prefixed to the result. This means that
- 15338 if the <space> and '+' flags both appear, the <space> flag shall be ignored.
- 15339 # Specifies that the value is to be converted to an alternative form. For o conversion, it
- 15340 increases the precision (if necessary) to force the first digit of the result to be 0. For x or
- 15341 X conversion specifiers, a non-zero result shall have 0x (or 0X) prefixed to it. For a, A, e,
- 15342 E, f, F, g, and G conversion specifiers, the result shall always contain a radix character,
- 15343 even if no digits follow it. Without this flag, a radix character appears in the result of
- 15344 these conversions only if a digit follows it. For g and G conversion specifiers, trailing
- 15345 zeros shall *not* be removed from the result as they normally are. For other conversion
- 15346 specifiers, the behavior is undefined.

15347	0	For d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversion specifiers, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the '0' and '-' flags both appear, the '0' flag shall be ignored. For d, i, o, u, x, and X conversion specifiers, if a precision is specified, the '0' flag shall be ignored. If the '0' and '-' flags both appear, the grouping wide characters are inserted before zero padding. For other conversions, the behavior is undefined.
15348		
15349		
15350		
15351		
15352		
15353		
15354		The length modifiers and their meanings are:
15355	hh	Specifies that a following d, i, o, u, x, or X conversion specifier applies to a signed char or unsigned char argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to signed char or unsigned char before printing); or that a following n conversion specifier applies to a pointer to a signed char argument.
15356		
15357		
15358		
15359		
15360	h	Specifies that a following d, i, o, u, x, or X conversion specifier applies to a short or unsigned short argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to short or unsigned short before printing); or that a following n conversion specifier applies to a pointer to a short argument.
15361		
15362		
15363		
15364		
15365	l (ell)	Specifies that a following d, i, o, u, x, or X conversion specifier applies to a long or unsigned long argument; that a following n conversion specifier applies to a pointer to a long argument; that a following c conversion specifier applies to a wint_t argument; that a following s conversion specifier applies to a pointer to a wchar_t argument; or has no effect on a following a, A, e, E, f, F, g, or G conversion specifier.
15366		
15367		
15368		
15369		
15370	ll (ell-ell)	
15371		Specifies that a following d, i, o, u, x, or X conversion specifier applies to a long long or unsigned long long argument; or that a following n conversion specifier applies to a pointer to a long long argument.
15372		
15373		
15374	j	Specifies that a following d, i, o, u, x, or X conversion specifier applies to an intmax_t or uintmax_t argument; or that a following n conversion specifier applies to a pointer to an intmax_t argument.
15375		
15376		
15377	z	Specifies that a following d, i, o, u, x, or X conversion specifier applies to a size_t or the corresponding signed integer type argument; or that a following n conversion specifier applies to a pointer to a signed integer type corresponding to a size_t argument.
15378		
15379		
15380	t	Specifies that a following d, i, o, u, x, or X conversion specifier applies to a ptrdiff_t or the corresponding unsigned type argument; or that a following n conversion specifier applies to a pointer to a ptrdiff_t argument.
15381		
15382		
15383	L	Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to a long double argument.
15384		
15385		If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.
15386		
15387		The conversion specifiers and their meanings are:
15388	d, i	The int argument shall be converted to a signed decimal in the style "[−]dddd". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision shall be 1. The result of converting zero with an explicit precision of zero shall be no wide characters.
15389		
15390		
15391		
15392		

15393	o	The unsigned argument shall be converted to unsigned octal format in the style "dddd". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision shall be 1. The result of converting zero with an explicit precision of zero shall be no wide characters.
15394		
15395		
15396		
15397		
15398	u	The unsigned argument shall be converted to unsigned decimal format in the style "dddd". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision shall be 1. The result of converting zero with an explicit precision of zero shall be no wide characters.
15399		
15400		
15401		
15402		
15403	x	The unsigned argument shall be converted to unsigned hexadecimal format in the style "dddd"; the letters "abcdef" are used. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision shall be 1. The result of converting zero with an explicit precision of zero shall be no wide characters.
15404		
15405		
15406		
15407		
15408	X	Equivalent to the x conversion specifier, except that letters "ABCDEF" are used instead of "abcdef".
15409		
15410	f, F	The double argument shall be converted to decimal notation in the style "[−]ddd.ddd", where the number of digits after the radix character shall be equal to the precision specification. If the precision is missing, it shall be taken as 6; if the precision is explicitly zero and no '#' flag is present, no radix character shall appear. If a radix character appears, at least one digit shall appear before it. The value shall be rounded in an implementation-defined manner to the appropriate number of digits.
15411		
15412		
15413		
15414		
15415		
15416		A double argument representing an infinity shall be converted in one of the styles "[−]inf" or "[−]infinity"; which style is implementation-defined. A double argument representing a NaN shall be converted in one of the styles "[−]nan" or "[−]nan(<i>n-char-sequence</i>)"; which style, and the meaning of any <i>n-char-sequence</i> , is implementation-defined. The F conversion specifier produces "INF", "INFINITY", or "NAN" instead of "inf", "infinity", or "nan", respectively.
15417		
15418		
15419		
15420		
15421		
15422	e, E	The double argument shall be converted in the style "[−]d.ddde±dd", where there shall be one digit before the radix character (which is non-zero if the argument is non-zero) and the number of digits after it shall be equal to the precision; if the precision is missing, it shall be taken as 6; if the precision is zero and no '#' flag is present, no radix character shall appear. The value shall be rounded in an implementation-defined manner to the appropriate number of digits. The E conversion wide character shall produce a number with 'E' instead of 'e' introducing the exponent. The exponent shall always contain at least two digits. If the value is zero, the exponent shall be zero.
15423		
15424		
15425		
15426		
15427		
15428		
15429		
15430		A double argument representing an infinity or NaN shall be converted in the style of an f or F conversion specifier.
15431		
15432	g, G	The double argument shall be converted in the style f or e (or in the style F or E in the case of a G conversion specifier), with the precision specifying the number of significant digits. If an explicit precision is zero, it shall be taken as 1. The style used depends on the value converted; style e (or E) shall be used only if the exponent resulting from such a conversion is less than −4 or greater than or equal to the precision. Trailing zeros shall be removed from the fractional portion of the result; a radix character shall appear only if it is followed by a digit.
15433		
15434		
15435		
15436		
15437		
15438		
15439		A double argument representing an infinity or NaN shall be converted in the style of an f or F conversion specifier.
15440		

15441	a, A	A double argument representing a floating-point number shall be converted in the style "[−]0xh.hhhhp±d", where there shall be one hexadecimal digit (which is non-zero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point wide character and the number of hexadecimal digits after it shall be equal to the precision; if the precision is missing and FLT_RADIX is a power of 2, then the precision shall be sufficient for an exact representation of the value; if the precision is missing and FLT_RADIX is not a power of 2, then the precision shall be sufficient to distinguish values of type double , except that trailing zeros may be omitted; if the precision is zero and the '#' flag is not specified, no decimal-point wide character shall appear. The letters "abcdef" are used for a conversion and the letters "ABCDEF" for A conversion. The A conversion specifier produces a number with 'X' and 'P' instead of 'x' and 'p'. The exponent shall always contain at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent shall be zero.
15442		
15443		
15444		
15445		
15446		
15447		
15448		
15449		
15450		
15451		
15452		
15453		
15454		
15455		A double argument representing an infinity or NaN shall be converted in the style of an f or F conversion specifier.
15456		
15457	c	If no l (ell) qualifier is present, the int argument shall be converted to a wide character as if by calling the <i>btowc()</i> function and the resulting wide character shall be written. Otherwise, the wint_t argument shall be converted to wchar_t , and written.
15458		
15459		
15460	s	If no l (ell) qualifier is present, the application shall ensure that the argument is a pointer to a character array containing a character sequence beginning in the initial shift state. Characters from the array shall be converted as if by repeated calls to the <i>mbrtowc()</i> function, with the conversion state described by an mbstate_t object initialized to zero before the first character is converted, and written up to (but not including) the terminating null wide character. If the precision is specified, no more than that many wide characters shall be written. If the precision is not specified, or is greater than the size of the array, the application shall ensure that the array contains a null wide character.
15461		
15462		
15463		
15464		
15465		
15466		
15467		
15468		
15469		If an l (ell) qualifier is present, the application shall ensure that the argument is a pointer to an array of type wchar_t . Wide characters from the array shall be written up to (but not including) a terminating null wide character. If no precision is specified, or is greater than the size of the array, the application shall ensure that the array contains a null wide character. If a precision is specified, no more than that many wide characters shall be written.
15470		
15471		
15472		
15473		
15474		
15475	p	The application shall ensure that the argument is a pointer to void . The value of the pointer shall be converted to a sequence of printable wide characters in an implementation-defined manner.
15476		
15477		
15478	n	The application shall ensure that the argument is a pointer to an integer into which is written the number of wide characters written to the output so far by this call to one of the <i>fwprintf()</i> functions. No argument shall be converted, but one shall be consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.
15479		
15480		
15481		
15482		
15483	XSI C	Equivalent to lc.
15484	XSI S	Equivalent to ls.
15485	%	Output a '%' wide character; no argument shall be converted. The entire conversion specification shall be %%.
15486		

- 15487 If a conversion specification does not match one of the above forms, the behavior is undefined.
- 15488 In no case does a nonexistent or small field width cause truncation of a field; if the result of a
15489 conversion is wider than the field width, the field shall be expanded to contain the conversion
15490 result. Characters generated by *fwprintf()* and *wprintf()* shall be printed as if *fputwc()* had been
15491 called.
- 15492 For *a* and *A* conversions, if *FLT_RADIX* is not a power of 2 and the result is not exactly
15493 representable in the given precision, the result should be one of the two adjacent numbers in
15494 hexadecimal floating style with the given precision, with the extra stipulation that the error
15495 should have a correct sign for the current rounding direction.
- 15496 For *e*, *E*, *f*, *F*, *g*, and *G* conversion specifiers, if the number of significant decimal digits is at most
15497 *DECIMAL_DIG*, then the result should be correctly rounded. If the number of significant
15498 decimal digits is more than *DECIMAL_DIG* but the source value is exactly representable with
15499 *DECIMAL_DIG* digits, then the result should be an exact representation with trailing zeros.
15500 Otherwise, the source value is bounded by two adjacent decimal strings $L < U$, both having
15501 *DECIMAL_DIG* significant digits; the value of the resultant decimal string D should satisfy $L \leq$
15502 $D \leq U$, with the extra stipulation that the error should have a correct sign for the current
15503 rounding direction.
- 15504 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the call to a
15505 successful execution of *fwprintf()* or *wprintf()* and the next successful completion of a call to
15506 *fflush()* or *fclose()* on the same stream, or a call to *exit()* or *abort()*.
- 15507 **RETURN VALUE**
- 15508 Upon successful completion, these functions shall return the number of wide characters
15509 transmitted, excluding the terminating null wide character in the case of *swprintf()*, or a negative
15510 CX value if an output error was encountered, and set *errno* to indicate the error.
- 15511 If *n* or more wide characters were requested to be written, *swprintf()* shall return a negative
15512 CX value, and set *errno* to indicate the error.
- 15513 **ERRORS**
- 15514 For the conditions under which *fwprintf()* and *wprintf()* fail and may fail, refer to *fputwc()*.
- 15515 In addition, all forms of *fwprintf()* may fail if:
- 15516 XSI [EILSEQ] A wide-character code that does not correspond to a valid character has been
15517 detected.
- 15518 XSI [EINVAL] There are insufficient arguments.
- 15519 In addition, *wprintf()* and *fwprintf()* may fail if:
- 15520 XSI [ENOMEM] Insufficient storage space is available.
- 15521 **EXAMPLES**
- 15522 To print the language-independent date and time format, the following statement could be used:
- 15523 `wprintf(format, weekday, month, day, hour, min);`
- 15524 For American usage, *format* could be a pointer to the wide-character string:
- 15525 `L"%s, %s %d, %d:%.2d\n"`
- 15526 producing the message:
- 15527 `Sunday, July 3, 10:02`
- 15528 whereas for German usage, *format* could be a pointer to the wide-character string:

15529 L"%1\$s, %3\$d. %2\$s, %4\$d:%5\$.2d\n"

15530 producing the message:

15531 Sonntag, 3. Juli, 10:02

15532 APPLICATION USAGE

15533 None.

15534 RATIONALE

15535 None.

15536 FUTURE DIRECTIONS

15537 None.

15538 SEE ALSO

15539 *btowc()*, *fputwc()*, *fwscanf()*, *mbrtowc()*, *setlocale()*, the Base Definitions volume of
15540 IEEE Std 1003.1-2001, Chapter 7, Locale, <stdio.h>, <wchar.h>

15541 CHANGE HISTORY

15542 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
15543 (E).

15544 Issue 6

15545 The Open Group Corrigendum U040/1 is applied to the RETURN VALUE section, describing
15546 the case if *n* or more wide characters are requested to be written using *swprintf()*.

15547 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

15548 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 15549 • The prototypes for *fwprintf()*, *swprintf()*, and *wprintf()* are updated.
- 15550 • The DESCRIPTION is updated.
- 15551 • The *hh*, *ll*, *j*, *t*, and *z* length modifiers are added.
- 15552 • The *a*, *A*, and *F* conversion characters are added.
- 15553 • XSI shading is removed from the description of character string representations of infinity
15554 and NaN floating-point values.

15555 The DESCRIPTION is updated to use the terms “conversion specifier” and “conversion
15556 specification” consistently.

15557 ISO/IEC 9899:1999 standard, Technical Corrigendum No. 1 is incorporated.

15558 **NAME**15559 `fwrite` — binary output15560 **SYNOPSIS**15561 `#include <stdio.h>`

```
15562        size_t fwrite(const void *restrict ptr, size_t size, size_t nitems,
15563                      FILE *restrict stream);
```

15564 **DESCRIPTION**

15565 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 15566 conflict between the requirements described here and the ISO C standard is unintentional. This
 15567 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

15568 The `fwrite()` function shall write, from the array pointed to by `ptr`, up to `nitems` elements whose
 15569 size is specified by `size`, to the stream pointed to by `stream`. For each object, `size` calls shall be
 15570 made to the `fputc()` function, taking the values (in order) from an array of **unsigned char** exactly
 15571 overlaying the object. The file-position indicator for the stream (if defined) shall be advanced by
 15572 the number of bytes successfully written. If an error occurs, the resulting value of the file-
 15573 position indicator for the stream is unspecified.

15574 CX The `st_ctime` and `st_mtime` fields of the file shall be marked for update between the successful
 15575 execution of `fwrite()` and the next successful completion of a call to `fflush()` or `fclose()` on the
 15576 same stream, or a call to `exit()` or `abort()`.

15577 **RETURN VALUE**

15578 The `fwrite()` function shall return the number of elements successfully written, which may be
 15579 less than `nitems` if a write error is encountered. If `size` or `nitems` is 0, `fwrite()` shall return 0 and the
 15580 state of the stream remains unchanged. Otherwise, if a write error occurs, the error indicator for
 15581 CX the stream shall be set, and `errno` shall be set to indicate the error.

15582 **ERRORS**15583 Refer to `fputc()`.15584 **EXAMPLES**

15585 None.

15586 **APPLICATION USAGE**

15587 Because of possible differences in element length and byte ordering, files written using `fwrite()`
 15588 are application-dependent, and possibly cannot be read using `fread()` by a different application
 15589 or by the same application on a different processor.

15590 **RATIONALE**

15591 None.

15592 **FUTURE DIRECTIONS**

15593 None.

15594 **SEE ALSO**

15595 `ferror()`, `fopen()`, `printf()`, `putc()`, `puts()`, `write()`, the Base Definitions volume of
 15596 IEEE Std 1003.1-2001, `<stdio.h>`

15597 **CHANGE HISTORY**

15598 First released in Issue 1. Derived from Issue 1 of the SVID.

15599 **Issue 6**

15600 Extensions beyond the ISO C standard are marked.

15601 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

15602

- The *fwrite()* prototype is updated.

15603

- The DESCRIPTION is updated to clarify how the data is written out using *fputc()*.

15604 **NAME**

15605 fwscanf, swscanf, wscanf — convert formatted wide-character input

15606 **SYNOPSIS**

15607 #include <stdio.h>

15608 #include <wchar.h>

15609 int fwscanf(FILE *restrict *stream*, const wchar_t *restrict *format*, ...);15610 int swscanf(const wchar_t *restrict *ws*,15611 const wchar_t *restrict *format*, ...);15612 int wscanf(const wchar_t *restrict *format*, ...);15613 **DESCRIPTION**

15614 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 15615 conflict between the requirements described here and the ISO C standard is unintentional. This
 15616 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

15617 The *fwscanf()* function shall read from the named input *stream*. The *wscanf()* function shall read
 15618 from the standard input stream *stdin*. The *swscanf()* function shall read from the wide-character
 15619 string *ws*. Each function reads wide characters, interprets them according to a format, and stores
 15620 the results in its arguments. Each expects, as arguments, a control wide-character string *format*
 15621 described below, and a set of *pointer* arguments indicating where the converted input should be
 15622 stored. The result is undefined if there are insufficient arguments for the format. If the *format* is
 15623 exhausted while arguments remain, the excess arguments are evaluated but are otherwise
 15624 ignored.

15625 XSI Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than
 15626 to the next unused argument. In this case, the conversion specifier wide character % (see below)
 15627 is replaced by the sequence "%n\$", where *n* is a decimal integer in the range [1,{NL_ARGMAX}].
 15628 This feature provides for the definition of *format* wide-character strings that select arguments in
 15629 an order appropriate to specific languages. In *format* wide-character strings containing the
 15630 "%n\$" form of conversion specifications, it is unspecified whether numbered arguments in the
 15631 argument list can be referenced from the *format* wide-character string more than once.

15632 The *format* can contain either form of a conversion specification—that is, % or "%n\$"—but the
 15633 two forms cannot normally be mixed within a single *format* wide-character string. The only
 15634 exception to this is that %% or %* can be mixed with the "%n\$" form. When numbered
 15635 argument specifications are used, specifying the *N*th argument requires that all the leading
 15636 arguments, from the first to the (*N*–1)th, are pointers.

15637 CX The *fwscanf()* function in all its forms allows for detection of a language-dependent radix
 15638 character in the input string, encoded as a wide-character value. The radix character is defined in
 15639 the program's locale (category *LC_NUMERIC*). In the POSIX locale, or in a locale where the
 15640 radix character is not defined, the radix character shall default to a period (' . ').

15641 The *format* is a wide-character string composed of zero or more directives. Each directive is
 15642 composed of one of the following: one or more white-space wide characters (<space>*s*, <tab>*s*,
 15643 <newline>*s*, <vertical-tab>*s*, or <form-feed>*s*); an ordinary wide character (neither '%' nor a
 15644 white-space character); or a conversion specification. Each conversion specification is introduced
 15645 XSI by a '%' or the sequence "%n\$" after which the following appear in sequence:

- 15646 • An optional assignment-suppressing character ' * '.
- 15647 • An optional non-zero decimal integer that specifies the maximum field width.
- 15648 • An optional length modifier that specifies the size of the receiving object.

15649 • A conversion specifier wide character that specifies the type of conversion to be applied. The
15650 valid conversion specifiers are described below.

15651 The *fwscanf()* functions shall execute each directive of the format in turn. If a directive fails, as
15652 detailed below, the function shall return. Failures are described as input failures (due to the
15653 unavailability of input bytes) or matching failures (due to inappropriate input).

15654 A directive composed of one or more white-space wide characters is executed by reading input
15655 until no more valid input can be read, or up to the first wide character which is not a white-
15656 space wide character, which remains unread.

15657 A directive that is an ordinary wide character shall be executed as follows. The next wide
15658 character is read from the input and compared with the wide character that comprises the
15659 directive; if the comparison shows that they are not equivalent, the directive shall fail, and the
15660 differing and subsequent wide characters remain unread. Similarly, if end-of-file, an encoding
15661 error, or a read error prevents a wide character from being read, the directive shall fail.

15662 A directive that is a conversion specification defines a set of matching input sequences, as
15663 described below for each conversion wide character. A conversion specification is executed in
15664 the following steps.

15665 Input white-space wide characters (as specified by *isspace()*) shall be skipped, unless the
15666 conversion specification includes a *['*, *c*, or *n* conversion specifier.

15667 An item shall be read from the input, unless the conversion specification includes an *n*
15668 conversion specifier wide character. An input item is defined as the longest sequence of input
15669 wide characters, not exceeding any specified field width, which is an initial subsequence of a
15670 matching sequence. The first wide character, if any, after the input item shall remain unread. If
15671 the length of the input item is zero, the execution of the conversion specification shall fail; this
15672 condition is a matching failure, unless end-of-file, an encoding error, or a read error prevented
15673 input from the stream, in which case it is an input failure.

15674 Except in the case of a *%* conversion specifier, the input item (or, in the case of a *%n* conversion
15675 specification, the count of input wide characters) shall be converted to a type appropriate to the
15676 conversion wide character. If the input item is not a matching sequence, the execution of the
15677 conversion specification shall fail; this condition is a matching failure. Unless assignment
15678 suppression was indicated by a *'*'*, the result of the conversion shall be placed in the object
15679 pointed to by the first argument following the *format* argument that has not already received a
15680 XSI conversion result if the conversion specification is introduced by *%*, or in the *n*th argument if
15681 introduced by the wide-character sequence *"%n\$"*. If this object does not have an appropriate
15682 type, or if the result of the conversion cannot be represented in the space provided, the behavior
15683 is undefined.

15684 The length modifiers and their meanings are:

15685 hh	Specifies that a following <i>d</i> , <i>i</i> , <i>o</i> , <i>u</i> , <i>x</i> , <i>X</i> , or <i>n</i> conversion specifier applies to an 15686 argument with type pointer to signed char or unsigned char .
15687 h	Specifies that a following <i>d</i> , <i>i</i> , <i>o</i> , <i>u</i> , <i>x</i> , <i>X</i> , or <i>n</i> conversion specifier applies to an 15688 argument with type pointer to short or unsigned short .
15689 l (ell)	Specifies that a following <i>d</i> , <i>i</i> , <i>o</i> , <i>u</i> , <i>x</i> , <i>X</i> , or <i>n</i> conversion specifier applies to an 15690 argument with type pointer to long or unsigned long ; that a following <i>a</i> , <i>A</i> , <i>e</i> , <i>E</i> , <i>f</i> , <i>F</i> , <i>g</i> , 15691 or <i>G</i> conversion specifier applies to an argument with type pointer to double ; or that a 15692 following <i>c</i> , <i>s</i> , or <i>['</i> conversion specifier applies to an argument with type pointer to 15693 wchar_t .

15694	ll (ell-ell)	
15695		Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an
15696		argument with type pointer to long long or unsigned long long .
15697	j	Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an
15698		argument with type pointer to intmax_t or uintmax_t .
15699	z	Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an
15700		argument with type pointer to size_t or the corresponding signed integer type.
15701	t	Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an
15702		argument with type pointer to ptrdiff_t or the corresponding unsigned type.
15703	L	Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to an
15704		argument with type pointer to long double .
15705		If a length modifier appears with any conversion specifier other than as specified above, the
15706		behavior is undefined.
15707		The following conversion specifier wide characters are valid:
15708	d	Matches an optionally signed decimal integer, whose format is the same as expected for
15709		the subject sequence of <i>wcstol()</i> with the value 10 for the <i>base</i> argument. In the absence
15710		of a size modifier, the application shall ensure that the corresponding argument is a
15711		pointer to int .
15712	i	Matches an optionally signed integer, whose format is the same as expected for the
15713		subject sequence of <i>wcstol()</i> with 0 for the <i>base</i> argument. In the absence of a size
15714		modifier, the application shall ensure that the corresponding argument is a pointer to
15715		int .
15716	o	Matches an optionally signed octal integer, whose format is the same as expected for
15717		the subject sequence of <i>wcstoul()</i> with the value 8 for the <i>base</i> argument. In the absence
15718		of a size modifier, the application shall ensure that the corresponding argument is a
15719		pointer to unsigned .
15720	u	Matches an optionally signed decimal integer, whose format is the same as expected for
15721		the subject sequence of <i>wcstoul()</i> with the value 10 for the <i>base</i> argument. In the absence
15722		of a size modifier, the application shall ensure that the corresponding argument is a
15723		pointer to unsigned .
15724	x	Matches an optionally signed hexadecimal integer, whose format is the same as
15725		expected for the subject sequence of <i>wcstoul()</i> with the value 16 for the <i>base</i> argument.
15726		In the absence of a size modifier, the application shall ensure that the corresponding
15727		argument is a pointer to unsigned .
15728	a, e, f, g	
15729		Matches an optionally signed floating-point number, infinity, or NaN whose format is
15730		the same as expected for the subject sequence of <i>wcstod()</i> . In the absence of a size
15731		modifier, the application shall ensure that the corresponding argument is a pointer to
15732		float .
15733		If the <i>fwprintf()</i> family of functions generates character string representations for
15734		infinity and NaN (a symbolic entity encoded in floating-point format) to support
15735		IEEE Std 754-1985, the <i>fwscanf()</i> family of functions shall recognize them as input.
15736	s	Matches a sequence of non white-space wide characters. If no l (ell) qualifier is present,
15737		characters from the input field shall be converted as if by repeated calls to the
15738		<i>wcrtomb()</i> function, with the conversion state described by an mbstate_t object

15739		initialized to zero before the first wide character is converted. The application shall
15740		ensure that the corresponding argument is a pointer to a character array large enough
15741		to accept the sequence and the terminating null character, which shall be added
15742		automatically.
15743		Otherwise, the application shall ensure that the corresponding argument is a pointer to
15744		an array of wchar_t large enough to accept the sequence and the terminating null wide
15745		character, which shall be added automatically.
15746	[Matches a non-empty sequence of wide characters from a set of expected wide
15747		characters (the <i>scanset</i>). If no l (ell) qualifier is present, wide characters from the input
15748		field shall be converted as if by repeated calls to the <i>wcrtomb()</i> function, with the
15749		conversion state described by an mbstate_t object initialized to zero before the first
15750		wide character is converted. The application shall ensure that the corresponding
15751		argument is a pointer to a character array large enough to accept the sequence and the
15752		terminating null character, which shall be added automatically.
15753		If an l (ell) qualifier is present, the application shall ensure that the corresponding
15754		argument is a pointer to an array of wchar_t large enough to accept the sequence and
15755		the terminating null wide character, which shall be added automatically.
15756		The conversion specification includes all subsequent wide characters in the <i>format</i>
15757		string up to and including the matching right square bracket (']'). The wide
15758		characters between the square brackets (the <i>scanlist</i>) comprise the scanset, unless the
15759		wide character after the left square bracket is a circumflex ('^'), in which case the
15760		scanset contains all wide characters that do not appear in the scanlist between the
15761		circumflex and the right square bracket. If the conversion specification begins with
15762		"[]" or "[^]", the right square bracket is included in the scanlist and the next right
15763		square bracket is the matching right square bracket that ends the conversion
15764		specification; otherwise, the first right square bracket is the one that ends the
15765		conversion specification. If a '-' is in the scanlist and is not the first wide character,
15766		nor the second where the first wide character is a '^', nor the last wide character, the
15767		behavior is implementation-defined.
15768	c	Matches a sequence of wide characters of exactly the number specified by the field
15769		width (1 if no field width is present in the conversion specification).
15770		If no l (ell) length modifier is present, characters from the input field shall be converted
15771		as if by repeated calls to the <i>wcrtomb()</i> function, with the conversion state described by
15772		an mbstate_t object initialized to zero before the first wide character is converted. The
15773		corresponding argument shall be a pointer to the initial element of a character array
15774		large enough to accept the sequence. No null character is added.
15775		If an l (ell) length modifier is present, the corresponding argument shall be a pointer to
15776		the initial element of an array of wchar_t large enough to accept the sequence. No null
15777		wide character is added.
15778		Otherwise, the application shall ensure that the corresponding argument is a pointer to
15779		an array of wchar_t large enough to accept the sequence. No null wide character is
15780		added.
15781	p	Matches an implementation-defined set of sequences, which shall be the same as the set
15782		of sequences that is produced by the %p conversion specification of the corresponding
15783		<i>fwprintf()</i> functions. The application shall ensure that the corresponding argument is a
15784		pointer to a pointer to void . The interpretation of the input item is implementation-
15785		defined. If the input item is a value converted earlier during the same program
15786		execution, the pointer that results shall compare equal to that value; otherwise, the

15787		behavior of the %p conversion is undefined.
15788	n	No input is consumed. The application shall ensure that the corresponding argument is a pointer to the integer into which is to be written the number of wide characters read from the input so far by this call to the <i>fwscanf()</i> functions. Execution of a %n conversion specification shall not increment the assignment count returned at the completion of execution of the function. No argument shall be converted, but one shall be consumed. If the conversion specification includes an assignment-suppressing wide character or a field width, the behavior is undefined.
15789		
15790		
15791		
15792		
15793		
15794		
15795	XSI	C Equivalent to lc.
15796	XSI	S Equivalent to ls.
15797	%	Matches a single '%' wide character; no conversion or assignment shall occur. The complete conversion specification shall be %%.
15798		
15799		If a conversion specification is invalid, the behavior is undefined.
15800		The conversion specifiers A, E, F, G, and X are also valid and shall be equivalent to, respectively, a, e, f, g, and x.
15801		
15802		If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any wide characters matching the current conversion specification (except for %n) have been read (other than leading white-space, where permitted), execution of the current conversion specification shall terminate with an input failure. Otherwise, unless execution of the current conversion specification is terminated with a matching failure, execution of the following conversion specification (if any) shall be terminated with an input failure.
15803		
15804		
15805		
15806		
15807		
15808		Reaching the end of the string in <i>swscanf()</i> shall be equivalent to encountering end-of-file for <i>fwscanf()</i> .
15809		
15810		If conversion terminates on a conflicting input, the offending input shall be left unread in the input. Any trailing white space (including <newline>) shall be left unread unless matched by a conversion specification. The success of literal matches and suppressed assignments is only directly determinable via the %n conversion specification.
15811		
15812		
15813		
15814		
15814	CX	The <i>fwscanf()</i> and <i>wscanf()</i> functions may mark the <i>st_atime</i> field of the file associated with <i>stream</i> for update. The <i>st_atime</i> field shall be marked for update by the first successful execution of <i>fgetc()</i> , <i>fgetwc()</i> , <i>fgets()</i> , <i>fgetws()</i> , <i>fread()</i> , <i>getc()</i> , <i>getwc()</i> , <i>getchar()</i> , <i>getwchar()</i> , <i>gets()</i> , <i>fscanf()</i> , or <i>fwscanf()</i> using <i>stream</i> that returns data not supplied by a prior call to <i>ungetc()</i> .
15815		
15816		
15817		
15818	RETURN VALUE	
15819		Upon successful completion, these functions shall return the number of successfully matched and assigned input items; this number can be zero in the event of an early matching failure. If the input ends before the first matching failure or conversion, EOF shall be returned. If a read error occurs, the error indicator for the stream is set, EOF shall be returned, and <i>errno</i> shall be set to indicate the error.
15820		
15821		
15822	CX	
15823		
15824	ERRORS	
15825		For the conditions under which the <i>fwscanf()</i> functions shall fail and may fail, refer to <i>fgetwc()</i> .
15826		In addition, <i>fwscanf()</i> may fail if:
15827	XSI	[EILSEQ] Input byte sequence does not form a valid character.
15828	XSI	[EINVAL] There are insufficient arguments.

15829 **EXAMPLES**

15830 The call:

```
15831 int i, n; float x; char name[50];
15832 n = wscanf(L"%d%f%s", &i, &x, name);
```

15833 with the input line:

15834 25 54.32E-1 Hamster

15835 assigns to *n* the value 3, to *i* the value 25, to *x* the value 5.432, and *name* contains the string
 15836 "Hamster".

15837 The call:

```
15838 int i; float x; char name[50];
15839 (void) wscanf(L"%2d%f*d %[0123456789]", &i, &x, name);
```

15840 with input:

15841 56789 0123 56a72

15842 assigns 56 to *i*, 789.0 to *x*, skips 0123, and places the string "56\0" in *name*. The next call to
 15843 *getchar()* shall return the character 'a'.

15844 **APPLICATION USAGE**

15845 In format strings containing the '%' form of conversion specifications, each argument in the
 15846 argument list is used exactly once.

15847 **RATIONALE**

15848 None.

15849 **FUTURE DIRECTIONS**

15850 None.

15851 **SEE ALSO**

15852 *getwc()*, *fwprintf()*, *setlocale()*, *wctod()*, *wcstol()*, *wcstoul()*, *wcrtomb()*, the Base Definitions
 15853 volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <langinfo.h>, <stdio.h>, <wchar.h>

15854 **CHANGE HISTORY**

15855 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
 15856 (E).

15857 **Issue 6**

15858 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

15859 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 15860 • The prototypes for *fwscanf()* and *swscanf()* are updated.
- 15861 • The DESCRIPTION is updated.
- 15862 • The hh, ll, j, t, and z length modifiers are added.
- 15863 • The a, A, and F conversion characters are added.

15864 The DESCRIPTION is updated to use the terms “conversion specifier” and “conversion
 15865 specification” consistently.

15866 **NAME**

15867 gai_strerror — address and name information error description

15868 **SYNOPSIS**

15869 #include <netdb.h>

15870 const char *gai_strerror(int *ecode*);15871 **DESCRIPTION**15872 The *gai_strerror()* function shall return a text string describing an error value for the *getaddrinfo()*
15873 and *getnameinfo()* functions listed in the <netdb.h> header.15874 When the *ecode* argument is one of the following values listed in the <netdb.h> header:

15875 [EAI_AGAIN]

15876 [EAI_BADFLAGS]

15877 [EAI_FAIL]

15878 [EAI_FAMILY]

15879 [EAI_MEMORY]

15880 [EAI_NONAME]

15881 [EAI_SERVICE]

15882 [EAI_SOCKTYPE]

15883 [EAI_SYSTEM]

15884 the function return value shall point to a string describing the error. If the argument is not one
15885 of those values, the function shall return a pointer to a string whose contents indicate an
15886 unknown error.15887 **RETURN VALUE**15888 Upon successful completion, *gai_strerror()* shall return a pointer to an implementation-defined
15889 string.15890 **ERRORS**

15891 No errors are defined.

15892 **EXAMPLES**

15893 None.

15894 **APPLICATION USAGE**

15895 None.

15896 **RATIONALE**

15897 None.

15898 **FUTURE DIRECTIONS**

15899 None.

15900 **SEE ALSO**15901 *getaddrinfo()*, the Base Definitions volume of IEEE Std 1003.1-2001, <netdb.h>15902 **CHANGE HISTORY**

15903 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

15904 The Open Group Base Resolution bwg2001-009 is applied, which changes the return type from
15905 **char *** to **const char ***. This is for coordination with the IPnG Working Group.

15906 **NAME**15907 gcvt — convert a floating-point number to a string (**LEGACY**)15908 **SYNOPSIS**

15909 XSI #include <stdlib.h>

15910 char *gcvt(double *value*, int *ndigit*, char **buf*);

15911

15912 **DESCRIPTION**15913 Refer to *ecvt()*.

15914 **NAME**

15915 getaddrinfo — get address information

15916 **SYNOPSIS**

15917 #include <sys/socket.h>

15918 #include <netdb.h>

```
15919       int getaddrinfo(const char *restrict nodename,
15920                       const char *restrict servname,
15921                       const struct addrinfo *restrict hints,
15922                       struct addrinfo **restrict res);
```

15923 **DESCRIPTION**15924 Refer to *freeaddrinfo()*.

15925 **NAME**

15926 getc — get a byte from a stream

15927 **SYNOPSIS**

15928 #include <stdio.h>

15929 int getc(FILE **stream*);

15930 **DESCRIPTION**

15931 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
15932 conflict between the requirements described here and the ISO C standard is unintentional. This
15933 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

15934 The *getc()* function shall be equivalent to *fgetc()*, except that if it is implemented as a macro it
15935 may evaluate *stream* more than once, so the argument should never be an expression with side
15936 effects.

15937 **RETURN VALUE**

15938 Refer to *fgetc()*.

15939 **ERRORS**

15940 Refer to *fgetc()*.

15941 **EXAMPLES**

15942 None.

15943 **APPLICATION USAGE**

15944 If the integer value returned by *getc()* is stored into a variable of type **char** and then compared
15945 against the integer constant EOF, the comparison may never succeed, because sign-extension of
15946 a variable of type **char** on widening to integer is implementation-defined.

15947 Since it may be implemented as a macro, *getc()* may treat incorrectly a *stream* argument with
15948 side effects. In particular, *getc(*f++)* does not necessarily work as expected. Therefore, use of this
15949 function should be preceded by "#undef getc" in such situations; *fgetc()* could also be used.

15950 **RATIONALE**

15951 None.

15952 **FUTURE DIRECTIONS**

15953 None.

15954 **SEE ALSO**

15955 *fgetc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

15956 **CHANGE HISTORY**

15957 First released in Issue 1. Derived from Issue 1 of the SVID.

15958 **NAME**

15959 `getc_unlocked`, `getchar_unlocked`, `putc_unlocked`, `putchar_unlocked` — stdio with explicit client
 15960 locking

15961 **SYNOPSIS**

```
15962 TSF      #include <stdio.h>

15963          int getc_unlocked(FILE *stream);
15964          int getchar_unlocked(void);
15965          int putc_unlocked(int c, FILE *stream);
15966          int putchar_unlocked(int c);
15967
```

15968 **DESCRIPTION**

15969 Versions of the functions `getc()`, `getchar()`, `putc()`, and `putchar()` respectively named
 15970 `getc_unlocked()`, `getchar_unlocked()`, `putc_unlocked()`, and `putchar_unlocked()` shall be provided
 15971 which are functionally equivalent to the original versions, with the exception that they are not
 15972 required to be implemented in a thread-safe manner. They may only safely be used within a
 15973 scope protected by `flockfile()` (or `ftrylockfile()`) and `funlockfile()`. These functions may safely be
 15974 used in a multi-threaded program if and only if they are called while the invoking thread owns
 15975 the (FILE *) object, as is the case after a successful call to the `flockfile()` or `ftrylockfile()` functions.

15976 **RETURN VALUE**

15977 See `getc()`, `getchar()`, `putc()`, and `putchar()`.

15978 **ERRORS**

15979 See `getc()`, `getchar()`, `putc()`, and `putchar()`.

15980 **EXAMPLES**

15981 None.

15982 **APPLICATION USAGE**

15983 Since they may be implemented as macros, `getc_unlocked()` and `putc_unlocked()` may treat
 15984 incorrectly a *stream* argument with side effects. In particular, `getc_unlocked(*f++)` and
 15985 `putc_unlocked(*f++)` do not necessarily work as expected. Therefore, use of these functions in
 15986 such situations should be preceded by the following statement as appropriate:

```
15987          #undef getc_unlocked
15988          #undef putc_unlocked
```

15989 **RATIONALE**

15990 Some I/O functions are typically implemented as macros for performance reasons (for example,
 15991 `putc()` and `getc()`). For safety, they need to be synchronized, but it is often too expensive to
 15992 synchronize on every character. Nevertheless, it was felt that the safety concerns were more
 15993 important; consequently, the `getc()`, `getchar()`, `putc()`, and `putchar()` functions are required to be
 15994 thread-safe. However, unlocked versions are also provided with names that clearly indicate the
 15995 unsafe nature of their operation but can be used to exploit their higher performance. These
 15996 unlocked versions can be safely used only within explicitly locked program regions, using
 15997 exported locking primitives. In particular, a sequence such as:

```
15998          flockfile(fileptr);
15999          putc_unlocked('1', fileptr);
16000          putc_unlocked('\n', fileptr);
16001          fprintf(fileptr, "Line 2\n");
16002          funlockfile(fileptr);
```

16003 is permissible, and results in the text sequence:

16004 1
16005 Line 2
16006 being printed without being interspersed with output from other threads.

16007 It would be wrong to have the standard names such as *getc()*, *putc()*, and so on, map to the
16008 “faster, but unsafe” rather than the “slower, but safe” versions. In either case, you would still
16009 want to inspect all uses of *getc()*, *putc()*, and so on, by hand when converting existing code.
16010 Choosing the safe bindings as the default, at least, results in correct code and maintains the
16011 “atomicity at the function” invariant. To do otherwise would introduce gratuitous
16012 synchronization errors into converted code. Other routines that modify the *stdio* (**FILE** *)
16013 structures or buffers are also safely synchronized.

16014 Note that there is no need for functions of the form *getc_locked()*, *putc_locked()*, and so on, since
16015 this is the functionality of *getc()*, *putc()*, *et al.* It would be inappropriate to use a feature test
16016 macro to switch a macro definition of *getc()* between *getc_locked()* and *getc_unlocked()*, since the
16017 ISO C standard requires an actual function to exist, a function whose behavior could not be
16018 changed by the feature test macro. Also, providing both the *xxx_locked()* and *xxx_unlocked()*
16019 forms leads to the confusion of whether the suffix describes the behavior of the function or the
16020 circumstances under which it should be used.

16021 Three additional routines, *flockfile()*, *ftrylockfile()*, and *funlockfile()* (which may be macros), are
16022 provided to allow the user to delineate a sequence of I/O statements that are executed
16023 synchronously.

16024 The *ungetc()* function is infrequently called relative to the other functions/macros so no
16025 unlocked variation is needed.

16026 **FUTURE DIRECTIONS**
16027 None.

16028 **SEE ALSO**
16029 *getc()*, *getchar()*, *putc()*, *putchar()*, the Base Definitions volume of IEEE Std 1003.1-2001,
16030 **<stdio.h>**

16031 **CHANGE HISTORY**
16032 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

16033 **Issue 6**
16034 These functions are marked as part of the Thread-Safe Functions option.

16035 The Open Group Corrigendum U030/2 is applied, adding APPLICATION USAGE describing
16036 how applications should be written to avoid the case when the functions are implemented as
16037 macros.

16038 **NAME**16039 `getchar` — get a byte from a stdin stream16040 **SYNOPSIS**16041 `#include <stdio.h>`16042 `int getchar(void);`16043 **DESCRIPTION**

16044 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
16045 conflict between the requirements described here and the ISO C standard is unintentional. This
16046 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

16047 The `getchar()` function shall be equivalent to `getc(stdin)`.16048 **RETURN VALUE**16049 Refer to `fgetc()`.16050 **ERRORS**16051 Refer to `fgetc()`.16052 **EXAMPLES**

16053 None.

16054 **APPLICATION USAGE**

16055 If the integer value returned by `getchar()` is stored into a variable of type **char** and then
16056 compared against the integer constant EOF, the comparison may never succeed, because sign-
16057 extension of a variable of type **char** on widening to integer is implementation-defined.

16058 **RATIONALE**

16059 None.

16060 **FUTURE DIRECTIONS**

16061 None.

16062 **SEE ALSO**16063 `getc()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdio.h>`16064 **CHANGE HISTORY**

16065 First released in Issue 1. Derived from Issue 1 of the SVID.

16066 NAME

16067 getchar_unlocked — stdio with explicit client locking

16068 SYNOPSIS

16069 TSF #include <stdio.h>

16070 int getchar_unlocked(void);

16071

16072 DESCRIPTION

16073 Refer to *getc_unlocked()*.

16074 **NAME**

16075 getcontext, setcontext — get and set current user context

16076 **SYNOPSIS**

16077 xSI #include <ucontext.h>

16078 int getcontext(ucontext_t *ucp);

16079 int setcontext(const ucontext_t *ucp);

16080

16081 **DESCRIPTION**

16082 The *getcontext()* function shall initialize the structure pointed to by *ucp* to the current user
 16083 context of the calling thread. The **ucontext_t** type that *ucp* points to defines the user context and
 16084 includes the contents of the calling thread's machine registers, the signal mask, and the current
 16085 execution stack.

16086 The *setcontext()* function shall restore the user context pointed to by *ucp*. A successful call to
 16087 *setcontext()* shall not return; program execution resumes at the point specified by the *ucp*
 16088 argument passed to *setcontext()*. The *ucp* argument should be created either by a prior call to
 16089 *getcontext()* or *makecontext()*, or by being passed as an argument to a signal handler. If the *ucp*
 16090 argument was created with *getcontext()*, program execution continues as if the corresponding
 16091 call of *getcontext()* had just returned. If the *ucp* argument was created with *makecontext()*,
 16092 program execution continues with the function passed to *makecontext()*. When that function
 16093 returns, the thread shall continue as if after a call to *setcontext()* with the *ucp* argument that was
 16094 input to *makecontext()*. If the *uc_link* member of the **ucontext_t** structure pointed to by the *ucp*
 16095 argument is equal to 0, then this context is the main context, and the thread shall exit when this
 16096 context returns. The effects of passing a *ucp* argument obtained from any other source are
 16097 unspecified.

16098 **RETURN VALUE**

16099 Upon successful completion, *setcontext()* shall not return and *getcontext()* shall return 0;
 16100 otherwise, a value of -1 shall be returned.

16101 **ERRORS**

16102 No errors are defined.

16103 **EXAMPLES**16104 Refer to *makecontext()*.16105 **APPLICATION USAGE**

16106 When a signal handler is executed, the current user context is saved and a new context is
 16107 created. If the thread leaves the signal handler via *longjmp()*, then it is unspecified whether the
 16108 context at the time of the corresponding *setjmp()* call is restored and thus whether future calls to
 16109 *getcontext()* provide an accurate representation of the current context, since the context restored
 16110 by *longjmp()* does not necessarily contain all the information that *setcontext()* requires. Signal
 16111 handlers should use *siglongjmp()* or *setcontext()* instead.

16112 Conforming applications should not modify or access the *uc_mcontext* member of **ucontext_t**. A
 16113 conforming application cannot assume that context includes any process-wide static data,
 16114 possibly including *errno*. Users manipulating contexts should take care to handle these
 16115 explicitly when required.

16116 Use of contexts to create alternate stacks is not defined by this volume of IEEE Std 1003.1-2001.

16117 **RATIONALE**

16118 None.

16119 **FUTURE DIRECTIONS**

16120 None.

16121 **SEE ALSO**

16122 *bsd_signal()*, *makecontext()*, *setcontext()*, *setjmp()*, *sigaction()*, *sigaltstack()*, *siglongjmp()*,
16123 *sigprocmask()*, *sigsetjmp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**ucontext.h**>

16124 **CHANGE HISTORY**

16125 First released in Issue 4, Version 2.

16126 **Issue 5**

16127 Moved from X/OPEN UNIX extension to BASE.

16128 The following sentence was removed from the DESCRIPTION: “If the *ucp* argument was passed
16129 to a signal handler, program execution continues with the program instruction following the
16130 instruction interrupted by the signal.”

16131 **NAME**

16132 getcwd — get the pathname of the current working directory

16133 **SYNOPSIS**

16134 #include <unistd.h>

16135 char *getcwd(char *buf, size_t size);

16136 **DESCRIPTION**

16137 The *getcwd()* function shall place an absolute pathname of the current working directory in the
 16138 array pointed to by *buf*, and return *buf*. The pathname copied to the array shall contain no
 16139 components that are symbolic links. The *size* argument is the size in bytes of the character array
 16140 pointed to by the *buf* argument. If *buf* is a null pointer, the behavior of *getcwd()* is unspecified.

16141 **RETURN VALUE**

16142 Upon successful completion, *getcwd()* shall return the *buf* argument. Otherwise, *getcwd()* shall
 16143 return a null pointer and set *errno* to indicate the error. The contents of the array pointed to by
 16144 *buf* are then undefined.

16145 **ERRORS**16146 The *getcwd()* function shall fail if:16147 [EINVAL] The *size* argument is 0.

16148 [ERANGE] The *size* argument is greater than 0, but is smaller than the length of the
 16149 pathname +1.

16150 The *getcwd()* function may fail if:

16151 [EACCES] Read or search permission was denied for a component of the pathname.

16152 [ENOMEM] Insufficient storage space is available.

16153 **EXAMPLES**16154 **Determining the Absolute Pathname of the Current Working Directory**

16155 The following example returns a pointer to an array that holds the absolute pathname of the
 16156 current working directory. The pointer is returned in the *ptr* variable, which points to the *buf*
 16157 array where the pathname is stored.

16158 #include <stdlib.h>

16159 #include <unistd.h>

16160 ...

16161 long size;

16162 char *buf;

16163 char *ptr;

16164 size = pathconf(".", _PC_PATH_MAX);

16165 if ((buf = (char *)malloc((size_t)size)) != NULL)

16166 ptr = getcwd(buf, (size_t)size);

16167 ...

16168 **APPLICATION USAGE**

16169 None.

16170 **RATIONALE**

16171 Since the maximum pathname length is arbitrary unless {PATH_MAX} is defined, an application
16172 generally cannot supply a *buf* with *size* {{PATH_MAX}+1}.

16173 Having *getcwd()* take no arguments and instead use the *malloc()* function to produce space for
16174 the returned argument was considered. The advantage is that *getcwd()* knows how big the
16175 working directory pathname is and can allocate an appropriate amount of space. But the
16176 programmer would have to use the *free()* function to free the resulting object, or each use of
16177 *getcwd()* would further reduce the available memory. Also, *malloc()* and *free()* are used nowhere
16178 else in this volume of IEEE Std 1003.1-2001. Finally, *getcwd()* is taken from the SVID where it has
16179 the two arguments used in this volume of IEEE Std 1003.1-2001.

16180 The older function *getwd()* was rejected for use in this context because it had only a buffer
16181 argument and no *size* argument, and thus had no way to prevent overwriting the buffer, except
16182 to depend on the programmer to provide a large enough buffer.

16183 On some implementations, if *buf* is a null pointer, *getcwd()* may obtain *size* bytes of memory
16184 using *malloc()*. In this case, the pointer returned by *getcwd()* may be used as the argument in a
16185 subsequent call to *free()*. Invoking *getcwd()* with *buf* as a null pointer is not recommended in
16186 conforming applications.

16187 If a program is operating in a directory where some (grand)parent directory does not permit
16188 reading, *getcwd()* may fail, as in most implementations it must read the directory to determine
16189 the name of the file. This can occur if search, but not read, permission is granted in an
16190 intermediate directory, or if the program is placed in that directory by some more privileged
16191 process (for example, login). Including the [EACCES] error condition makes the reporting of the
16192 error consistent and warns the application writer that *getcwd()* can fail for reasons beyond the
16193 control of the application writer or user. Some implementations can avoid this occurrence (for
16194 example, by implementing *getcwd()* using *pwd*, where *pwd* is a set-user-root process), thus the
16195 error was made optional. Since this volume of IEEE Std 1003.1-2001 permits the addition of other
16196 errors, this would be a common addition and yet one that applications could not be expected to
16197 deal with without this addition.

16198 **FUTURE DIRECTIONS**

16199 None.

16200 **SEE ALSO**

16201 *malloc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

16202 **CHANGE HISTORY**

16203 First released in Issue 1. Derived from Issue 1 of the SVID.

16204 **Issue 6**

16205 The following new requirements on POSIX implementations derive from alignment with the
16206 Single UNIX Specification:

- 16207 • The [ENOMEM] optional error condition is added.

16208 **NAME**

16209 getdate — convert user format date and time

16210 **SYNOPSIS**

16211 XSI #include <time.h>

16212 struct tm *getdate(const char *string);

16213

16214 **DESCRIPTION**16215 The *getdate()* function shall convert a string representation of a date or time into a broken-down
16216 time.16217 The external variable or macro *getdate_err* is used by *getdate()* to return error values.16218 Templates are used to parse and interpret the input string. The templates are contained in a text
16219 file identified by the environment variable *DATMSK*. The *DATMSK* variable should be set to
16220 indicate the full pathname of the file that contains the templates. The first line in the template
16221 that matches the input specification is used for interpretation and conversion into the internal
16222 time format.

16223 The following conversion specifications shall be supported:

16224 %% Equivalent to %.

16225 %a Abbreviated weekday name.

16226 %A Full weekday name.

16227 %b Abbreviated month name.

16228 %B Full month name.

16229 %c Locale's appropriate date and time representation.

16230 %C Century number [00,99]; leading zeros are permitted but not required.

16231 %d Day of month [01,31]; the leading 0 is optional.

16232 %D Date as %m/%d/%y.

16233 %e Equivalent to %d.

16234 %h Abbreviated month name.

16235 %H Hour [00,23].

16236 %I Hour [01,12].

16237 %m Month number [01,12].

16238 %M Minute [00,59].

16239 %n Equivalent to <newline>.

16240 %p Locale's equivalent of either AM or PM.

16241 %p The locale's appropriate representation of time in AM and PM notation. In the POSIX
16242 locale, this shall be equivalent to %I:%M:%S %p.

16243 %R Time as %H:%M.

16244 %S Seconds [00,60]. The range goes to 60 (rather than stopping at 59) to allow positive leap
16245 seconds to be expressed. Since leap seconds cannot be predicted by any algorithm, leap
16246 second data must come from some external source.

16247	%t	Equivalent to <tab>.
16248	%T	Time as %H:%M:%S.
16249	%w	Weekday number (Sunday = [0,6]).
16250	%x	Locale's appropriate date representation.
16251	%X	Locale's appropriate time representation.
16252	%Y	Year within century. When a century is not otherwise specified, values in the range [69,99] shall refer to years 1969 to 1999 inclusive, and values in the range [00,68] shall refer to years 2000 to 2068 inclusive.
16253		
16254		
16255		Note: It is expected that in a future version of IEEE Std 1003.1-2001 the default century inferred from a 2-digit year will change. (This would apply to all commands accepting a 2-digit year as input.)
16256		
16257		
16258	%Y	Year as "ccyy" (for example, 2001).
16259	%Z	Timezone name or no characters if no timezone exists. If the timezone supplied by %Z is not the timezone that <i>getdate()</i> expects, an invalid input specification error shall result. The <i>getdate()</i> function calculates an expected timezone based on information supplied to the function (such as the hour, day, and month).
16260		
16261		
16262		
16263		The match between the template and input specification performed by <i>getdate()</i> shall be case-insensitive.
16264		
16265		The month and weekday names can consist of any combination of upper and lowercase letters. The process can request that the input date or time specification be in a specific language by setting the <i>LC_TIME</i> category (see <i>setlocale()</i>).
16266		
16267		
16268		Leading zeros are not necessary for the descriptors that allow leading zeros. However, at most two digits are allowed for those descriptors, including leading zeros. Extra whitespace in either the template file or in <i>string</i> shall be ignored.
16269		
16270		
16271		The results are undefined if the conversion specifications %c, %x, and %X include unsupported conversion specifications.
16272		
16273		The following rules apply for converting the input specification into the internal format:
16274		<ul style="list-style-type: none"> • If %Z is being scanned, then <i>getdate()</i> shall initialize the broken-down time to be the current time in the scanned timezone. Otherwise, it shall initialize the broken-down time based on the current local time as if <i>localtime()</i> had been called.
16275		
16276		
16277		<ul style="list-style-type: none"> • If only the weekday is given, the day chosen shall be the day, starting with today and moving into the future, which first matches the named day.
16278		
16279		<ul style="list-style-type: none"> • If only the month (and no year) is given, the month chosen shall be the month, starting with the current month and moving into the future, which first matches the named month. The first day of the month shall be assumed if no day is given.
16280		
16281		
16282		<ul style="list-style-type: none"> • If no hour, minute, and second are given, the current hour, minute, and second shall be assumed.
16283		
16284		<ul style="list-style-type: none"> • If no date is given, the hour chosen shall be the hour, starting with the current hour and moving into the future, which first matches the named hour.
16285		
16286		If a conversion specification in the <i>DATMSK</i> file does not correspond to one of the conversion specifications above, the behavior is unspecified.
16287		
16288		The <i>getdate()</i> function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.
16289		

16290 **RETURN VALUE**

16291 Upon successful completion, *getdate()* shall return a pointer to a **struct tm**. Otherwise, it shall
 16292 return a null pointer and set *getdate_err* to indicate the error.

16293 **ERRORS**

16294 The *getdate()* function shall fail in the following cases, setting *getdate_err* to the value shown in
 16295 the list below. Any changes to *errno* are unspecified.

- 16296 1. The *DATMSK* environment variable is null or undefined.
- 16297 2. The template file cannot be opened for reading.
- 16298 3. Failed to get file status information.
- 16299 4. The template file is not a regular file.
- 16300 5. An I/O error is encountered while reading the template file.
- 16301 6. Memory allocation failed (not enough memory available).
- 16302 7. There is no line in the template that matches the input.
- 16303 8. Invalid input specification. For example, February 31; or a time is specified that cannot be
 16304 represented in a **time_t** (representing the time in seconds since the Epoch).

16305 **EXAMPLES**

- 16306 1. The following example shows the possible contents of a template:

```
16307 %m
16308 %A %B %d, %Y, %H:%M:%S
16309 %A
16310 %B
16311 %m/%d/%Y %I %p
16312 %d,%m,%Y %H:%M
16313 at %A the %dst of %B in %Y
16314 run job at %I %p,%B %dnd
16315 %A den %d. %B %Y %H.%M Uhr
```

- 16316 2. The following are examples of valid input specifications for the template in Example 1:

```
16317 getdate("10/1/87 4 PM");
16318 getdate("Friday");
16319 getdate("Friday September 18, 1987, 10:30:30");
16320 getdate("24,9,1986 10:30");
16321 getdate("at monday the 1st of december in 1986");
16322 getdate("run job at 3 PM, december 2nd");
```

16323 If the *LC_TIME* category is set to a German locale that includes *freitag* as a weekday name
 16324 and *oktober* as a month name, the following would be valid:

```
16325 getdate("freitag den 10. oktober 1986 10.30 Uhr");
```

- 16326 3. The following example shows how local date and time specification can be defined in the
 16327 template:

16328
16329
16330
16331
16332
16333

Invocation	Line in Template
getdate("11/27/86")	%m/%d/%y
getdate("27.11.86")	%d.%m.%y
getdate("86-11-27")	%y-%m-%d
getdate("Friday 12:00:00")	%A %H:%M:%S

16334
16335
16336
16337

4. The following examples help to illustrate the above rules assuming that the current date is Mon Sep 22 12:19:47 EDT 1986 and the *LC_TIME* category is set to the default C locale:

16338
16339
16340
16341
16342
16343
16344
16345
16346
16347
16348
16349
16350
16351

Input	Line in Template	Date
Mon	%a	Mon Sep 22 12:19:47 EDT 1986
Sun	%a	Sun Sep 28 12:19:47 EDT 1986
Fri	%a	Fri Sep 26 12:19:47 EDT 1986
September	%B	Mon Sep 1 12:19:47 EDT 1986
January	%B	Thu Jan 1 12:19:47 EST 1987
December	%B	Mon Dec 1 12:19:47 EST 1986
Sep Mon	%b %a	Mon Sep 1 12:19:47 EDT 1986
Jan Fri	%b %a	Fri Jan 2 12:19:47 EST 1987
Dec Mon	%b %a	Mon Dec 1 12:19:47 EST 1986
Jan Wed 1989	%b %a %Y	Wed Jan 4 12:19:47 EST 1989
Fri 9	%a %H	Fri Sep 26 09:00:00 EDT 1986
Feb 10:30	%b %H:%S	Sun Feb 1 10:00:30 EST 1987
10:30	%H:%M	Tue Sep 23 10:30:00 EDT 1986
13:30	%H:%M	Mon Sep 22 13:30:00 EDT 1986

16352 APPLICATION USAGE

16353 Although historical versions of *getdate()* did not require that **<time.h>** declare the external
16354 variable *getdate_err*, this volume of IEEE Std 1003.1-2001 does require it. The standard
16355 developers encourage applications to remove declarations of *getdate_err* and instead incorporate
16356 the declaration by including **<time.h>**.

16357 Applications should use %Y (4-digit years) in preference to %y (2-digit years).

16358 RATIONALE

16359 In standard locales, the conversion specifications %c, %x, and %X do not include unsupported
16360 conversion specifiers and so the text regarding results being undefined is not a problem in that
16361 case.

16362 FUTURE DIRECTIONS

16363 None.

16364 SEE ALSO

16365 *ctime()*, *localtime()*, *setlocale()*, *strftime()*, *times()*, the Base Definitions volume of
16366 IEEE Std 1003.1-2001, **<time.h>**

16367 CHANGE HISTORY

16368 First released in Issue 4, Version 2.

16369 Issue 5

16370 Moved from X/OPEN UNIX extension to BASE.

16371 The last paragraph of the DESCRIPTION is added.

16372 The %C conversion specification is added, and the exact meaning of the %y conversion
16373 specification is clarified in the DESCRIPTION.

- 16374 A note indicating that this function need not be reentrant is added to the DESCRIPTION.
- 16375 The %R conversion specification is changed to follow historical practice.
- 16376 **Issue 6**
- 16377 The DESCRIPTION is updated to refer to “seconds since the Epoch” rather than “seconds since
- 16378 00:00:00 UTC (Coordinated Universal Time), January 1 1970” for consistency with other *time*
- 16379 functions.
- 16380 The description of %S is updated so that the valid range is [00,60] rather than [00,61].
- 16381 The DESCRIPTION is updated to refer to conversion specifications instead of field descriptors
- 16382 for consistency with other functions.

16383 **NAME**

16384 getegid — get the effective group ID

16385 **SYNOPSIS**

16386 #include <unistd.h>

16387 gid_t getegid(void);

16388 **DESCRIPTION**

16389 The *getegid()* function shall return the effective group ID of the calling process.

16390 **RETURN VALUE**

16391 The *getegid()* function shall always be successful and no return value is reserved to indicate an
16392 error.

16393 **ERRORS**

16394 No errors are defined.

16395 **EXAMPLES**

16396 None.

16397 **APPLICATION USAGE**

16398 None.

16399 **RATIONALE**

16400 None.

16401 **FUTURE DIRECTIONS**

16402 None.

16403 **SEE ALSO**

16404 *geteuid()*, *getgid()*, *getuid()*, *setegid()*, *seteuid()*, *setgid()*, *setregid()*, *setreuid()*, *setuid()*, the Base
16405 Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

16406 **CHANGE HISTORY**

16407 First released in Issue 1. Derived from Issue 1 of the SVID.

16408 **Issue 6**

16409 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

16410 The following new requirements on POSIX implementations derive from alignment with the
16411 Single UNIX Specification:

- 16412
 - The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
16413 required for conforming implementations of previous POSIX specifications, it was not
16414 required for UNIX applications.

16415 **NAME**

16416 getenv — get value of an environment variable

16417 **SYNOPSIS**

16418 #include <stdlib.h>

16419 char *getenv(const char *name);

16420 **DESCRIPTION**

16421 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 16422 conflict between the requirements described here and the ISO C standard is unintentional. This
 16423 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

16424 The *getenv()* function shall search the environment of the calling process (see the Base
 16425 Definitions volume of IEEE Std 1003.1-2001, Chapter 8, Environment Variables) for the
 16426 environment variable *name* if it exists and return a pointer to the value of the environment
 16427 variable. If the specified environment variable cannot be found, a null pointer shall be returned.
 16428 The application shall ensure that it does not modify the string pointed to by the *getenv()*
 16429 function.

16430 CX The string pointed to may be overwritten by a subsequent call to *getenv()*, *setenv()*, or *unsetenv()*,
 16431 but shall not be overwritten by a call to any other function in this volume of
 16432 IEEE Std 1003.1-2001.

16433 CX If the application modifies *environ* or the pointers to which it points, the behavior of *getenv()* is
 16434 undefined.

16435 The *getenv()* function need not be reentrant. A function that is not required to be reentrant is not
 16436 required to be thread-safe.

16437 **RETURN VALUE**

16438 Upon successful completion, *getenv()* shall return a pointer to a string containing the *value* for
 16439 the specified *name*. If the specified *name* cannot be found in the environment of the calling
 16440 process, a null pointer shall be returned.

16441 The return value from *getenv()* may point to static data which may be overwritten by
 16442 CX subsequent calls to *getenv()*, *setenv()*, or *unsetenv()*.

16443 XSI On XSI-conformant systems, the return value from *getenv()* may point to static data which may
 16444 also be overwritten by subsequent calls to *putenv()*.

16445 **ERRORS**

16446 No errors are defined.

16447 **EXAMPLES**16448 **Getting the Value of an Environment Variable**16449 The following example gets the value of the *HOME* environment variable.

16450 #include <stdlib.h>

16451 ...

16452 const char *name = "HOME";

16453 char *value;

16454 value = getenv(name);

16455 **APPLICATION USAGE**

16456 None.

16457 **RATIONALE**

16458 The *clearenv()* function was considered but rejected. The *putenv()* function has now been
16459 included for alignment with the Single UNIX Specification.

16460 The *getenv()* function is inherently not reentrant because it returns a value pointing to static
16461 data.

16462 Conforming applications are required not to modify *environ* directly, but to use only the
16463 functions described here to manipulate the process environment as an abstract object. Thus, the
16464 implementation of the environment access functions has complete control over the data
16465 structure used to represent the environment (subject to the requirement that *environ* be
16466 maintained as a list of strings with embedded equal signs for applications that wish to scan the
16467 environment). This constraint allows the implementation to properly manage the memory it
16468 allocates, either by using allocated storage for all variables (copying them on the first invocation
16469 of *setenv()* or *unsetenv()*), or keeping track of which strings are currently in allocated space and
16470 which are not, via a separate table or some other means. This enables the implementation to free
16471 any allocated space used by strings (and perhaps the pointers to them) stored in *environ* when
16472 *unsetenv()* is called. A C runtime start-up procedure (that which invokes *main()* and perhaps
16473 initializes *environ*) can also initialize a flag indicating that none of the environment has yet been
16474 copied to allocated storage, or that the separate table has not yet been initialized.

16475 In fact, for higher performance of *getenv()*, the implementation could also maintain a separate
16476 copy of the environment in a data structure that could be searched much more quickly (such as
16477 an indexed hash table, or a binary tree), and update both it and the linear list at *environ* when
16478 *setenv()* or *unsetenv()* is invoked.

16479 Performance of *getenv()* can be important for applications which have large numbers of
16480 environment variables. Typically, applications like this use the environment as a resource
16481 database of user-configurable parameters. The fact that these variables are in the user's shell
16482 environment usually means that any other program that uses environment variables (such as *ls*,
16483 which attempts to use *COLUMNS*), or really almost any utility (*LANG*, *LC_ALL*, and so on) is
16484 similarly slowed down by the linear search through the variables.

16485 An implementation that maintains separate data structures, or even one that manages the
16486 memory it consumes, is not currently required as it was thought it would reduce consensus
16487 among implementors who do not want to change their historical implementations.

16488 The POSIX Threads Extension states that multi-threaded applications must not modify *environ*
16489 directly, and that IEEE Std 1003.1-2001 is providing functions which such applications can use in
16490 the future to manipulate the environment in a thread-safe manner. Thus, moving away from
16491 application use of *environ* is desirable from that standpoint as well.

16492 **FUTURE DIRECTIONS**

16493 None.

16494 **SEE ALSO**

16495 *exec*, *putenv()*, *setenv()*, *unsetenv()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter
16496 8, Environment Variables, <stdlib.h>

16497 **CHANGE HISTORY**

16498 First released in Issue 1. Derived from Issue 1 of the SVID.

16499 Issue 5

16500 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
16501 VALUE section.

16502 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

16503 Issue 6

16504 The following changes were made to align with the IEEE P1003.1a draft standard:

- 16505 • References added to the new *setenv()* and *unsetenv()* functions.

16506 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

16507 **NAME**

16508 geteuid — get the effective user ID

16509 **SYNOPSIS**

16510 #include <unistd.h>

16511 uid_t geteuid(void);

16512 **DESCRIPTION**

16513 The *geteuid()* function shall return the effective user ID of the calling process.

16514 **RETURN VALUE**

16515 The *geteuid()* function shall always be successful and no return value is reserved to indicate an
16516 error.

16517 **ERRORS**

16518 No errors are defined.

16519 **EXAMPLES**

16520 None.

16521 **APPLICATION USAGE**

16522 None.

16523 **RATIONALE**

16524 None.

16525 **FUTURE DIRECTIONS**

16526 None.

16527 **SEE ALSO**

16528 *getegid()*, *getgid()*, *getuid()*, *setegid()*, *seteuid()*, *setgid()*, *setregid()*, *setreuid()*, *setuid()*, the Base
16529 Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

16530 **CHANGE HISTORY**

16531 First released in Issue 1. Derived from Issue 1 of the SVID.

16532 **Issue 6**

16533 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

16534 The following new requirements on POSIX implementations derive from alignment with the
16535 Single UNIX Specification:

- 16536
 - The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
16537 required for conforming implementations of previous POSIX specifications, it was not
16538 required for UNIX applications.

16539 **NAME**

16540 getgid — get the real group ID

16541 **SYNOPSIS**

16542 #include <unistd.h>

16543 gid_t getgid(void);

16544 **DESCRIPTION**

16545 The *getgid()* function shall return the real group ID of the calling process.

16546 **RETURN VALUE**

16547 The *getgid()* function shall always be successful and no return value is reserved to indicate an
16548 error.

16549 **ERRORS**

16550 No errors are defined.

16551 **EXAMPLES**

16552 None.

16553 **APPLICATION USAGE**

16554 None.

16555 **RATIONALE**

16556 None.

16557 **FUTURE DIRECTIONS**

16558 None.

16559 **SEE ALSO**

16560 *getegid()*, *geteuid()*, *getuid()*, *setegid()*, *seteuid()*, *setgid()*, *setregid()*, *setreuid()*, *setuid()*, the Base
16561 Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

16562 **CHANGE HISTORY**

16563 First released in Issue 1. Derived from Issue 1 of the SVID.

16564 **Issue 6**

16565 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

16566 The following new requirements on POSIX implementations derive from alignment with the
16567 Single UNIX Specification:

- 16568
 - The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
16569 required for conforming implementations of previous POSIX specifications, it was not
16570 required for UNIX applications.

16571 NAME

16572 getgrent — get the group database entry

16573 SYNOPSIS

16574 xSI #include <grp.h>

16575 struct group *getgrent(void);

16576

16577 DESCRIPTION

16578 Refer to *endgrent()*.

16579 NAME

16580 getgrgid, getgrgid_r — get group database entry for a group ID

16581 SYNOPSIS

16582 #include <grp.h>

16583 struct group *getgrgid(gid_t gid);

16584 TSF int getgrgid_r(gid_t gid, struct group *grp, char *buffer,

16585 size_t bufsize, struct group **result);

16586

16587 DESCRIPTION

16588 The *getgrgid()* function shall search the group database for an entry with a matching *gid*.16589 The *getgrgid()* function need not be reentrant. A function that is not required to be reentrant is
16590 not required to be thread-safe.16591 TSF The *getgrgid_r()* function shall update the **group** structure pointed to by *grp* and store a pointer
16592 to that structure at the location pointed to by *result*. The structure shall contain an entry from
16593 the group database with a matching *gid*. Storage referenced by the group structure is allocated
16594 from the memory provided with the *buffer* parameter, which is *bufsize* bytes in size. The
16595 maximum size needed for this buffer can be determined with the {_SC_GETGR_R_SIZE_MAX}
16596 *sysconf()* parameter. A NULL pointer shall be returned at the location pointed to by *result* on
16597 error or if the requested entry is not found.

16598 RETURN VALUE

16599 Upon successful completion, *getgrgid()* shall return a pointer to a **struct group** with the structure
16600 defined in <grp.h> with a matching entry if one is found. The *getgrgid()* function shall return a
16601 null pointer if either the requested entry was not found, or an error occurred. On error, *errno*
16602 shall be set to indicate the error.16603 The return value may point to a static area which is overwritten by a subsequent call to
16604 *getgrent()*, *getgrgid()*, or *getgrnam()*.16605 TSF If successful, the *getgrgid_r()* function shall return zero; otherwise, an error number shall be
16606 returned to indicate the error.

16607 ERRORS

16608 The *getgrgid()* and *getgrgid_r()* functions may fail if:

16609 [EIO] An I/O error has occurred.

16610 [EINTR] A signal was caught during *getgrgid()*.

16611 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

16612 [ENFILE] The maximum allowable number of files is currently open in the system.

16613 TSF The *getgrgid_r()* function may fail if:16614 TSF [ERANGE] Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to
16615 be referenced by the resulting **group** structure.

16616 **EXAMPLES**16617 **Finding an Entry in the Group Database**

16618 The following example uses *getgrgid()* to search the group database for a group ID that was
 16619 previously stored in a **stat** structure, then prints out the group name if it is found. If the group is
 16620 not found, the program prints the numeric value of the group for the entry.

```
16621 #include <sys/types.h>
16622 #include <grp.h>
16623 #include <stdio.h>
16624 ...
16625 struct stat statbuf;
16626 struct group *grp;
16627 ...
16628 if ((grp = getgrgid(statbuf.st_gid)) != NULL)
16629     printf(" %-8.8s", grp->gr_name);
16630 else
16631     printf(" %-8d", statbuf.st_gid);
16632 ...
```

16633 **APPLICATION USAGE**

16634 Applications wishing to check for error situations should set *errno* to 0 before calling *getgrgid()*.
 16635 If *errno* is set on return, an error occurred.

16636 The *getgrgid_r()* function is thread-safe and shall return values in a user-supplied buffer instead
 16637 of possibly using a static data area that may be overwritten by each call.

16638 **RATIONALE**

16639 None.

16640 **FUTURE DIRECTIONS**

16641 None.

16642 **SEE ALSO**

16643 *endgrent()*, *getgrnam()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<grp.h>**,
 16644 **<limits.h>**, **<sys/types.h>**

16645 **CHANGE HISTORY**

16646 First released in Issue 1. Derived from System V Release 2.0.

16647 **Issue 5**

16648 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
 16649 VALUE section.

16650 The *getgrgid_r()* function is included for alignment with the POSIX Threads Extension.

16651 A note indicating that the *getgrgid()* function need not be reentrant is added to the
 16652 DESCRIPTION.

16653 **Issue 6**

16654 The *getgrgid_r()* function is marked as part of the Thread-Safe Functions option.

16655 The Open Group Corrigendum U028/3 is applied, correcting text in the DESCRIPTION
 16656 describing matching the *gid*.

16657 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

16658 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

- 16659 The following new requirements on POSIX implementations derive from alignment with the
16660 Single UNIX Specification:
- 16661 • The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was
16662 required for conforming implementations of previous POSIX specifications, it was not
16663 required for UNIX applications.
 - 16664 • In the RETURN VALUE section, the requirement to set *errno* on error is added.
 - 16665 • The [EIO], [EINTR], [EMFILE], and [ENFILE] optional error conditions are added.
- 16666 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
16667 its avoidance of possibly using a static data area.
- 16668 IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the
16669 buffer from *bufsize* characters to bytes.

16670 NAME

16671 getgrnam, getgrnam_r — search group database for a name

16672 SYNOPSIS

16673 #include <grp.h>

16674 struct group *getgrnam(const char *name);

16675 TSF int getgrnam_r(const char *name, struct group *grp, char *buffer,

16676 size_t bufsize, struct group **result);

16677

16678 DESCRIPTION

16679 The *getgrnam()* function shall search the group database for an entry with a matching *name*.16680 The *getgrnam()* function need not be reentrant. A function that is not required to be reentrant is
16681 not required to be thread-safe.16682 TSF The *getgrnam_r()* function shall update the **group** structure pointed to by *grp* and store a pointer
16683 to that structure at the location pointed to by *result*. The structure shall contain an entry from
16684 the group database with a matching *gid* or *name*. Storage referenced by the **group** structure is
16685 allocated from the memory provided with the *buffer* parameter, which is *bufsize* bytes in size. The
16686 maximum size needed for this buffer can be determined with the {_SC_GETGR_R_SIZE_MAX}
16687 *sysconf()* parameter. A NULL pointer is returned at the location pointed to by *result* on error or if
16688 the requested entry is not found.

16689 RETURN VALUE

16690 The *getgrnam()* function shall return a pointer to a **struct group** with the structure defined in
16691 <grp.h> with a matching entry if one is found. The *getgrnam()* function shall return a null
16692 pointer if either the requested entry was not found, or an error occurred. On error, *errno* shall be
16693 set to indicate the error.16694 The return value may point to a static area which is overwritten by a subsequent call to
16695 *getgrent()*, *getgrgid()*, or *getgrnam()*.16696 TSF If successful, the *getgrnam_r()* function shall return zero; otherwise, an error number shall be
16697 returned to indicate the error.

16698 ERRORS

16699 The *getgrnam()* and *getgrnam_r()* functions may fail if:

16700 [EIO] An I/O error has occurred.

16701 [EINTR] A signal was caught during *getgrnam()*.

16702 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

16703 [ENFILE] The maximum allowable number of files is currently open in the system.

16704 The *getgrnam_r()* function may fail if:16705 TSF [ERANGE] Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to
16706 be referenced by the resulting **group** structure.

16707 **EXAMPLES**

16708 None.

16709 **APPLICATION USAGE**

16710 Applications wishing to check for error situations should set *errno* to 0 before calling *getgrnam()*.
 16711 If *errno* is set on return, an error occurred.

16712 The *getgrnam_r()* function is thread-safe and shall return values in a user-supplied buffer instead
 16713 of possibly using a static data area that may be overwritten by each call.

16714 **RATIONALE**

16715 None.

16716 **FUTURE DIRECTIONS**

16717 None.

16718 **SEE ALSO**

16719 *endgrent()*, *getgrgid()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<grp.h>**, **<limits.h>**,
 16720 **<sys/types.h>**

16721 **CHANGE HISTORY**

16722 First released in Issue 1. Derived from System V Release 2.0.

16723 **Issue 5**

16724 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
 16725 VALUE section.

16726 The *getgrnam_r()* function is included for alignment with the POSIX Threads Extension.

16727 A note indicating that the *getgrnam()* function need not be reentrant is added to the
 16728 DESCRIPTION.

16729 **Issue 6**16730 The *getgrnam_r()* function is marked as part of the Thread-Safe Functions option.

16731 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

16732 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

16733 The following new requirements on POSIX implementations derive from alignment with the
 16734 Single UNIX Specification:

16735 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
 16736 required for conforming implementations of previous POSIX specifications, it was not
 16737 required for UNIX applications.

16738 • In the RETURN VALUE section, the requirement to set *errno* on error is added.

16739 • The [EIO], [EINTR], [EMFILE], and [ENFILE] optional error conditions are added.

16740 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
 16741 its avoidance of possibly using a static data area.

16742 IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the
 16743 buffer from *bufsize* characters to bytes.

16744 **NAME**

16745 getgroups — get supplementary group IDs

16746 **SYNOPSIS**

16747 #include <unistd.h>

16748 int getgroups(int *gidsetsize*, gid_t *grouplist*[]);16749 **DESCRIPTION**

16750 The *getgroups()* function shall fill in the array *grouplist* with the current supplementary group
 16751 IDs of the calling process. It is implementation-defined whether *getgroups()* also returns the
 16752 effective group ID in the *grouplist* array.

16753 The *gidsetsize* argument specifies the number of elements in the array *grouplist*. The actual
 16754 number of group IDs stored in the array shall be returned. The values of array entries with
 16755 indices greater than or equal to the value returned are undefined.

16756 If *gidsetsize* is 0, *getgroups()* shall return the number of group IDs that it would otherwise return
 16757 without modifying the array pointed to by *grouplist*.

16758 If the effective group ID of the process is returned with the supplementary group IDs, the value
 16759 returned shall always be greater than or equal to one and less than or equal to the value of
 16760 {NGROUPS_MAX}+1.

16761 **RETURN VALUE**

16762 Upon successful completion, the number of supplementary group IDs shall be returned. A
 16763 return value of -1 indicates failure and *errno* shall be set to indicate the error.

16764 **ERRORS**16765 The *getgroups()* function shall fail if:

16766	[EINVAL]	The <i>gidsetsize</i> argument is non-zero and less than the number of group IDs
16767		that would have been returned.

16768 **EXAMPLES**16769 **Getting the Supplementary Group IDs of the Calling Process**

16770 The following example places the current supplementary group IDs of the calling process into
 16771 the *group* array.

```

16772       #include <sys/types.h>
16773       #include <unistd.h>
16774       ...
16775       gid_t *group;
16776       int nogroups;
16777       long ngroups_max;

16778       ngroups_max = sysconf(_SC_NGROUPS_MAX) + 1;
16779       group = (gid_t *)malloc(ngroups_max * sizeof(gid_t));

16780       ngroups = getgroups(ngroups_max, group);

```

16781 **APPLICATION USAGE**

16782 None.

16783 **RATIONALE**

16784 The related function *setgroups()* is a privileged operation and therefore is not covered by this
 16785 volume of IEEE Std 1003.1-2001.

16786 As implied by the definition of supplementary groups, the effective group ID may appear in the
16787 array returned by *getgroups()* or it may be returned only by *getegid()*. Duplication may exist, but
16788 the application needs to call *getegid()* to be sure of getting all of the information. Various
16789 implementation variations and administrative sequences cause the set of groups appearing in
16790 the result of *getgroups()* to vary in order and as to whether the effective group ID is included,
16791 even when the set of groups is the same (in the mathematical sense of “set”). (The history of a
16792 process and its parents could affect the details of the result.)

16793 Application writers should note that {NGROUPS_MAX} is not necessarily a constant on all
16794 implementations.

16795 FUTURE DIRECTIONS

16796 None.

16797 SEE ALSO

16798 *getegid()*, *setgid()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>,
16799 <unistd.h>

16800 CHANGE HISTORY

16801 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

16802 Issue 5

16803 Normative text previously in the APPLICATION USAGE section is moved to the
16804 DESCRIPTION.

16805 Issue 6

16806 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

16807 The following new requirements on POSIX implementations derive from alignment with the
16808 Single UNIX Specification:

- 16809 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
16810 required for conforming implementations of previous POSIX specifications, it was not
16811 required for UNIX applications.
- 16812 • A return value of 0 is not permitted, because {NGROUPS_MAX} cannot be 0. This is a FIPS
16813 requirement.

16814 The following changes were made to align with the IEEE P1003.1a draft standard:

- 16815 • Explanation added that the effective group ID may be included in the supplementary group
16816 list.

16817 NAME

16818 gethostbyaddr, gethostbyname — network host database functions

16819 SYNOPSIS

16820 #include <netdb.h>

```
16821 OB struct hostent *gethostbyaddr(const void *addr, socklen_t len,
16822 int type);
16823 struct hostent *gethostbyname(const char *name);
16824
```

16825 DESCRIPTION

16826 These functions shall retrieve information about hosts. This information is considered to be
 16827 stored in a database that can be accessed sequentially or randomly. Implementation of this
 16828 database is unspecified.

16829 **Note:** In many cases it is implemented by the Domain Name System, as documented in RFC 1034,
 16830 RFC 1035, and RFC 1886.

16831 Entries shall be returned in **hostent** structures.

16832 The *gethostbyaddr()* function shall return an entry containing addresses of address family *type* for
 16833 the host with address *addr*. The *len* argument contains the length of the address pointed to by
 16834 *addr*. The *gethostbyaddr()* function need not be reentrant. A function that is not required to be
 16835 reentrant is not required to be thread-safe.

16836 The *gethostbyname()* function shall return an entry containing addresses of address family
 16837 AF_INET for the host with name *name*. The *gethostbyname()* function need not be reentrant. A
 16838 function that is not required to be reentrant is not required to be thread-safe.

16839 The *addr* argument of *gethostbyaddr()* shall be an **in_addr** structure when *type* is AF_INET. It
 16840 contains a binary format (that is, not null-terminated) address in network byte order. The
 16841 *gethostbyaddr()* function is not guaranteed to return addresses of address families other than
 16842 AF_INET, even when such addresses exist in the database.

16843 If *gethostbyaddr()* returns successfully, then the *h_addrtype* field in the result shall be the same as
 16844 the *type* argument that was passed to the function, and the *h_addr_list* field shall list a single
 16845 address that is a copy of the *addr* argument that was passed to the function.

16846 The *name* argument of *gethostbyname()* shall be a node name; the behavior of *gethostbyname()*
 16847 when passed a numeric address string is unspecified. For IPv4, a numeric address string shall be
 16848 in the dotted-decimal notation described in *inet_addr()*.

16849 If *name* is not a numeric address string and is an alias for a valid host name, then *gethostbyname()*
 16850 shall return information about the host name to which the alias refers, and *name* shall be
 16851 included in the list of aliases returned.

16852 RETURN VALUE

16853 Upon successful completion, these functions shall return a pointer to a **hostent** structure if the
 16854 requested entry was found, and a null pointer if the end of the database was reached or the
 16855 requested entry was not found.

16856 Upon unsuccessful completion, *gethostbyaddr()* and *gethostbyname()* shall set *h_errno* to indicate
 16857 the error.

16858 ERRORS

16859 These functions shall fail in the following cases. The *gethostbyaddr()* and *gethostbyname()*
 16860 functions shall set *h_errno* to the value shown in the list below. Any changes to *errno* are
 16861 unspecified.

16862 [HOST_NOT_FOUND]
16863 No such host is known.

16864 [NO_DATA] The server recognized the request and the name, but no address is available.
16865 Another type of request to the name server for the domain might return an
16866 answer.

16867 [NO_RECOVERY]
16868 An unexpected server failure occurred which cannot be recovered.

16869 [TRY_AGAIN] A temporary and possibly transient error occurred, such as a failure of a
16870 server to respond.

16871 EXAMPLES

16872 None.

16873 APPLICATION USAGE

16874 The *gethostbyaddr()* and *gethostbyname()* functions may return pointers to static data, which may
16875 be overwritten by subsequent calls to any of these functions.

16876 The *getaddrinfo()* and *getnameinfo()* functions are preferred over the *gethostbyaddr()* and
16877 *gethostbyname()* functions.

16878 RATIONALE

16879 None.

16880 FUTURE DIRECTIONS

16881 The *gethostbyaddr()* and *gethostbyname()* functions may be withdrawn in a future version.

16882 SEE ALSO

16883 *endhostent()*, *endservent()*, *gai_strerror()*, *getaddrinfo()*, *h_errno*, *inet_addr()*, the Base Definitions
16884 volume of IEEE Std 1003.1-2001, <netdb.h>

16885 CHANGE HISTORY

16886 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

16887 NAME

16888 gethostent — network host database functions

16889 SYNOPSIS

16890 #include <netdb.h>

16891 struct hostent *gethostent(void);

16892 DESCRIPTION

16893 Refer to *endhostent()*.

16894 **NAME**

16895 gethostid — get an identifier for the current host

16896 **SYNOPSIS**

16897 XSI #include <unistd.h>

16898 long gethostid(void);

16899

16900 **DESCRIPTION**

16901 The *gethostid()* function shall retrieve a 32-bit identifier for the current host.

16902 **RETURN VALUE**

16903 Upon successful completion, *gethostid()* shall return an identifier for the current host.

16904 **ERRORS**

16905 No errors are defined.

16906 **EXAMPLES**

16907 None.

16908 **APPLICATION USAGE**

16909 This volume of IEEE Std 1003.1-2001 does not define the domain in which the return value is
16910 unique.

16911 **RATIONALE**

16912 None.

16913 **FUTURE DIRECTIONS**

16914 None.

16915 **SEE ALSO**

16916 *random()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

16917 **CHANGE HISTORY**

16918 First released in Issue 4, Version 2.

16919 **Issue 5**

16920 Moved from X/OPEN UNIX extension to BASE.

16921 **NAME**

16922 gethostname — get name of current host

16923 **SYNOPSIS**

16924 #include <unistd.h>

16925 int gethostname(char *name, size_t namelen);

16926 **DESCRIPTION**

16927 The *gethostname()* function shall return the standard host name for the current machine. The
16928 *namelen* argument shall specify the size of the array pointed to by the *name* argument. The
16929 returned name shall be null-terminated, except that if *namelen* is an insufficient length to hold
16930 the host name, then the returned name shall be truncated and it is unspecified whether the
16931 returned name is null-terminated.

16932 Host names are limited to {HOST_NAME_MAX} bytes.

16933 **RETURN VALUE**

16934 Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned.

16935 **ERRORS**

16936 No errors are defined.

16937 **EXAMPLES**

16938 None.

16939 **APPLICATION USAGE**

16940 None.

16941 **RATIONALE**

16942 None.

16943 **FUTURE DIRECTIONS**

16944 None.

16945 **SEE ALSO**16946 *gethostid()*, *uname()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>16947 **CHANGE HISTORY**

16948 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

16949 The Open Group Base Resolution bwg2001-008 is applied, changing the *namelen* parameter from
16950 *socklen_t* to *size_t*.

16951 **NAME**

16952 getitimer, setitimer — get and set value of interval timer

16953 **SYNOPSIS**

16954 XSI #include <sys/time.h>

16955 int getitimer(int which, struct itimerval *value);

16956 int setitimer(int which, const struct itimerval *restrict value,

16957 struct itimerval *restrict ovalue);

16958

16959 **DESCRIPTION**

16960 The *getitimer()* function shall store the current value of the timer specified by *which* into the
 16961 structure pointed to by *value*. The *setitimer()* function shall set the timer specified by *which* to
 16962 the value specified in the structure pointed to by *value*, and if *ovalue* is not a null pointer, store
 16963 the previous value of the timer in the structure pointed to by *ovalue*.

16964 A timer value is defined by the **itimerval** structure, specified in <sys/time.h>. If *it_value* is non-
 16965 zero, it shall indicate the time to the next timer expiration. If *it_interval* is non-zero, it shall
 16966 specify a value to be used in reloading *it_value* when the timer expires. Setting *it_value* to 0 shall
 16967 disable a timer, regardless of the value of *it_interval*. Setting *it_interval* to 0 shall disable a timer
 16968 after its next expiration (assuming *it_value* is non-zero).

16969 Implementations may place limitations on the granularity of timer values. For each interval
 16970 timer, if the requested timer value requires a finer granularity than the implementation supports,
 16971 the actual timer value shall be rounded up to the next supported value.

16972 An XSI-conforming implementation provides each process with at least three interval timers,
 16973 which are indicated by the *which* argument:

16974 ITIMER_REAL Decrements in real time. A SIGALRM signal is delivered when this timer
 16975 expires.

16976 ITIMER_VIRTUAL Decrements in process virtual time. It runs only when the process is
 16977 executing. A SIGVTALRM signal is delivered when it expires.

16978 ITIMER_PROF Decrements both in process virtual time and when the system is running
 16979 on behalf of the process. It is designed to be used by interpreters in
 16980 statistically profiling the execution of interpreted programs. Each time the
 16981 ITIMER_PROF timer expires, the SIGPROF signal is delivered.

16982 The interaction between *setitimer()* and any of *alarm()*, *sleep()*, or *usleep()* is unspecified.

16983 **RETURN VALUE**

16984 Upon successful completion, *getitimer()* or *setitimer()* shall return 0; otherwise, -1 shall be
 16985 returned and *errno* set to indicate the error.

16986 **ERRORS**

16987 The *setitimer()* function shall fail if:

16988 [EINVAL] The *value* argument is not in canonical form. (In canonical form, the number of
 16989 microseconds is a non-negative integer less than 1 000 000 and the number of
 16990 seconds is a non-negative integer.)

16991 The *getitimer()* and *setitimer()* functions may fail if:

16992 [EINVAL] The *which* argument is not recognized.

16993 EXAMPLES

16994 None.

16995 APPLICATION USAGE

16996 None.

16997 RATIONALE

16998 None.

16999 FUTURE DIRECTIONS

17000 None.

17001 SEE ALSO

17002 *alarm()*, *sleep()*, *timer_getoverrun()*, *ualarm()*, *usleep()*, the Base Definitions volume of
17003 IEEE Std 1003.1-2001, <**signal.h**>, <**sys/time.h**>

17004 CHANGE HISTORY

17005 First released in Issue 4, Version 2.

17006 Issue 5

17007 Moved from X/OPEN UNIX extension to BASE.

17008 Issue 6

17009 The **restrict** keyword is added to the *setitimer()* prototype for alignment with the
17010 ISO/IEC 9899:1999 standard.

17011 **NAME**

17012 getlogin, getlogin_r — get login name

17013 **SYNOPSIS**

17014 #include <unistd.h>

17015 char *getlogin(void);

17016 TSF int getlogin_r(char *name, size_t namesize);

17017

17018 **DESCRIPTION**

17019 The *getlogin()* function shall return a pointer to a string containing the user name associated by
 17020 the login activity with the controlling terminal of the current process. If *getlogin()* returns a non-
 17021 null pointer, then that pointer points to the name that the user logged in under, even if there are
 17022 several login names with the same user ID.

17023 The *getlogin()* function need not be reentrant. A function that is not required to be reentrant is
 17024 not required to be thread-safe.

17025 TSF The *getlogin_r()* function shall put the name associated by the login activity with the controlling
 17026 terminal of the current process in the character array pointed to by *name*. The array is *namesize*
 17027 characters long and should have space for the name and the terminating null character. The
 17028 maximum size of the login name is {LOGIN_NAME_MAX}.

17029 If *getlogin_r()* is successful, *name* points to the name the user used at login, even if there are
 17030 several login names with the same user ID.

17031 **RETURN VALUE**

17032 Upon successful completion, *getlogin()* shall return a pointer to the login name or a null pointer
 17033 if the user's login name cannot be found. Otherwise, it shall return a null pointer and set *errno* to
 17034 indicate the error.

17035 The return value from *getlogin()* may point to static data whose content is overwritten by each
 17036 call.

17037 TSF If successful, the *getlogin_r()* function shall return zero; otherwise, an error number shall be
 17038 returned to indicate the error.

17039 **ERRORS**

17040 The *getlogin()* and *getlogin_r()* functions may fail if:

17041 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

17042 [ENFILE] The maximum allowable number of files is currently open in the system.

17043 [ENXIO] The calling process has no controlling terminal.

17044 The *getlogin_r()* function may fail if:

17045 TSF [ERANGE] The value of *namesize* is smaller than the length of the string to be returned
 17046 including the terminating null character.

17047 **EXAMPLES**17048 **Getting the User Login Name**

17049 The following example calls the *getlogin()* function to obtain the name of the user associated
 17050 with the calling process, and passes this information to the *getpwnam()* function to get the
 17051 associated user database information.

```

17052 #include <unistd.h>
17053 #include <sys/types.h>
17054 #include <pwd.h>
17055 #include <stdio.h>
17056 ...
17057 char *lgn;
17058 struct passwd *pw;
17059 ...
17060 if ((lgn = getlogin()) == NULL || (pw = getpwnam(lgn)) == NULL) {
17061     fprintf(stderr, "Get of user information failed.\n"); exit(1);
17062 }
```

17063 **APPLICATION USAGE**

17064 Three names associated with the current process can be determined: *getpwuid(geteuid())* shall
 17065 return the name associated with the effective user ID of the process; *getlogin()* shall return the
 17066 name associated with the current login activity; and *getpwuid(getuid())* shall return the name
 17067 associated with the real user ID of the process.

17068 The *getlogin_r()* function is thread-safe and returns values in a user-supplied buffer instead of
 17069 possibly using a static data area that may be overwritten by each call.

17070 **RATIONALE**

17071 The *getlogin()* function returns a pointer to the user's login name. The same user ID may be
 17072 shared by several login names. If it is desired to get the user database entry that is used during
 17073 login, the result of *getlogin()* should be used to provide the argument to the *getpwnam()*
 17074 function. (This might be used to determine the user's login shell, particularly where a single user
 17075 has multiple login shells with distinct login names, but the same user ID.)

17076 The information provided by the *cuserid()* function, which was originally defined in the
 17077 POSIX.1-1988 standard and subsequently removed, can be obtained by the following:

```
17078 getpwuid(geteuid())
```

17079 while the information provided by historical implementations of *cuserid()* can be obtained by:

```
17080 getpwuid(getuid())
```

17081 The thread-safe version of this function places the user name in a user-supplied buffer and
 17082 returns a non-zero value if it fails. The non-thread-safe version may return the name in a static
 17083 data area that may be overwritten by each call.

17084 **FUTURE DIRECTIONS**

17085 None.

17086 **SEE ALSO**

17087 *getpwnam()*, *getpwuid()*, *geteuid()*, *getuid()*, the Base Definitions volume of IEEE Std 1003.1-2001,
 17088 <limits.h>, <unistd.h>

17089 **CHANGE HISTORY**

17090 First released in Issue 1. Derived from System V Release 2.0.

17091 **Issue 5**

17092 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
17093 VALUE section.

17094 The *getlogin_r()* function is included for alignment with the POSIX Threads Extension.

17095 A note indicating that the *getlogin()* function need not be reentrant is added to the
17096 DESCRIPTION.

17097 **Issue 6**

17098 The *getlogin_r()* function is marked as part of the Thread-Safe Functions option.

17099 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

17100 The following new requirements on POSIX implementations derive from alignment with the
17101 Single UNIX Specification:

17102 • In the RETURN VALUE section, the requirement to set *errno* on error is added.

17103 • The [EMFILE], [ENFILE], and [ENXIO] optional error conditions are added.

17104 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
17105 its avoidance of possibly using a static data area.

17106 NAME

17107 getmsg, getpmsg — receive next message from a STREAMS file (STREAMS)

17108 SYNOPSIS

17109 XSR `#include <stropts.h>`

```

17110 int getmsg(int fildes, struct strbuf *restrict ctlptr,
17111           struct strbuf *restrict dataptr, int *restrict flagsp);
17112 int getpmsg(int fildes, struct strbuf *restrict ctlptr,
17113            struct strbuf *restrict dataptr, int *restrict bandp,
17114            int *restrict flagsp);
17115

```

17116 DESCRIPTION

17117 The *getmsg()* function shall retrieve the contents of a message located at the head of the
 17118 STREAM head read queue associated with a STREAMS file and place the contents into one or
 17119 more buffers. The message contains either a data part, a control part, or both. The data and
 17120 control parts of the message shall be placed into separate buffers, as described below. The
 17121 semantics of each part are defined by the originator of the message.

17122 The *getpmsg()* function shall be equivalent to *getmsg()*, except that it provides finer control over
 17123 the priority of the messages received. Except where noted, all requirements on *getmsg()* also
 17124 pertain to *getpmsg()*.

17125 The *fildes* argument specifies a file descriptor referencing a STREAMS-based file.

17126 The *ctlptr* and *dataptr* arguments each point to a **strbuf** structure, in which the *buf* member points
 17127 to a buffer in which the data or control information is to be placed, and the *maxlen* member
 17128 indicates the maximum number of bytes this buffer can hold. On return, the *len* member shall
 17129 contain the number of bytes of data or control information actually received. The *len* member
 17130 shall be set to 0 if there is a zero-length control or data part and *len* shall be set to -1 if no data or
 17131 control information is present in the message.

17132 When *getmsg()* is called, *flagsp* should point to an integer that indicates the type of message the
 17133 process is able to receive. This is described further below.

17134 The *ctlptr* argument is used to hold the control part of the message, and *dataptr* is used to hold
 17135 the data part of the message. If *ctlptr* (or *dataptr*) is a null pointer or the *maxlen* member is -1, the
 17136 control (or data) part of the message shall not be processed and shall be left on the STREAM
 17137 head read queue, and if the *ctlptr* (or *dataptr*) is not a null pointer, *len* shall be set to -1. If the
 17138 *maxlen* member is set to 0 and there is a zero-length control (or data) part, that zero-length part
 17139 shall be removed from the read queue and *len* shall be set to 0. If the *maxlen* member is set to 0
 17140 and there are more than 0 bytes of control (or data) information, that information shall be left on
 17141 the read queue and *len* shall be set to 0. If the *maxlen* member in *ctlptr* (or *dataptr*) is less than the
 17142 control (or data) part of the message, *maxlen* bytes shall be retrieved. In this case, the remainder
 17143 of the message shall be left on the STREAM head read queue and a non-zero return value shall
 17144 be provided.

17145 By default, *getmsg()* shall process the first available message on the STREAM head read queue.
 17146 However, a process may choose to retrieve only high-priority messages by setting the integer
 17147 pointed to by *flagsp* to RS_HIPRI. In this case, *getmsg()* shall only process the next message if it is
 17148 a high-priority message. When the integer pointed to by *flagsp* is 0, any available message shall
 17149 be retrieved. In this case, on return, the integer pointed to by *flagsp* shall be set to RS_HIPRI if a
 17150 high-priority message was retrieved, or 0 otherwise.

17151 For *getpmsg()*, the flags are different. The *flagsp* argument points to a bitmask with the following
 17152 mutually-exclusive flags defined: MSG_HIPRI, MSG_BAND, and MSG_ANY. Like *getmsg()*,

17153 *getpmsg()* shall process the first available message on the STREAM head read queue. A process
 17154 may choose to retrieve only high-priority messages by setting the integer pointed to by *flagsp* to
 17155 MSG_HIPRI and the integer pointed to by *bandp* to 0. In this case, *getpmsg()* shall only process
 17156 the next message if it is a high-priority message. In a similar manner, a process may choose to
 17157 retrieve a message from a particular priority band by setting the integer pointed to by *flagsp* to
 17158 MSG_BAND and the integer pointed to by *bandp* to the priority band of interest. In this case,
 17159 *getpmsg()* shall only process the next message if it is in a priority band equal to, or greater than,
 17160 the integer pointed to by *bandp*, or if it is a high-priority message. If a process wants to get the
 17161 first message off the queue, the integer pointed to by *flagsp* should be set to MSG_ANY and the
 17162 integer pointed to by *bandp* should be set to 0. On return, if the message retrieved was a high-
 17163 priority message, the integer pointed to by *flagsp* shall be set to MSG_HIPRI and the integer
 17164 pointed to by *bandp* shall be set to 0. Otherwise, the integer pointed to by *flagsp* shall be set to
 17165 MSG_BAND and the integer pointed to by *bandp* shall be set to the priority band of the message.

17166 If O_NONBLOCK is not set, *getmsg()* and *getpmsg()* shall block until a message of the type
 17167 specified by *flagsp* is available at the front of the STREAM head read queue. If O_NONBLOCK is
 17168 set and a message of the specified type is not present at the front of the read queue, *getmsg()* and
 17169 *getpmsg()* shall fail and set *errno* to [EAGAIN].

17170 If a hangup occurs on the STREAM from which messages are retrieved, *getmsg()* and *getpmsg()*
 17171 shall continue to operate normally, as described above, until the STREAM head read queue is
 17172 empty. Thereafter, they shall return 0 in the *len* members of *ctlptr* and *dataptr*.

17173 RETURN VALUE

17174 Upon successful completion, *getmsg()* and *getpmsg()* shall return a non-negative value. A value
 17175 of 0 indicates that a full message was read successfully. A return value of MORECTL indicates
 17176 that more control information is waiting for retrieval. A return value of MOREDATA indicates
 17177 that more data is waiting for retrieval. A return value of the bitwise-logical OR of MORECTL
 17178 and MOREDATA indicates that both types of information remain. Subsequent *getmsg()* and
 17179 *getpmsg()* calls shall retrieve the remainder of the message. However, if a message of higher
 17180 priority has come in on the STREAM head read queue, the next call to *getmsg()* or *getpmsg()*
 17181 shall retrieve that higher-priority message before retrieving the remainder of the previous
 17182 message.

17183 If the high priority control part of the message is consumed, the message shall be placed back on
 17184 the queue as a normal message of band 0. Subsequent *getmsg()* and *getpmsg()* calls shall retrieve
 17185 the remainder of the message. If, however, a priority message arrives or already exists on the
 17186 STREAM head, the subsequent call to *getmsg()* or *getpmsg()* shall retrieve the higher-priority
 17187 message before retrieving the remainder of the message that was put back.

17188 Upon failure, *getmsg()* and *getpmsg()* shall return -1 and set *errno* to indicate the error.

17189 ERRORS

17190 The *getmsg()* and *getpmsg()* functions shall fail if:

- | | | |
|-------|-----------|--|
| 17191 | [EAGAIN] | The O_NONBLOCK flag is set and no messages are available. |
| 17192 | [EBADF] | The <i>fildes</i> argument is not a valid file descriptor open for reading. |
| 17193 | [EBADMSG] | The queued message to be read is not valid for <i>getmsg()</i> or <i>getpmsg()</i> or a
17194 pending file descriptor is at the STREAM head. |
| 17195 | [EINTR] | A signal was caught during <i>getmsg()</i> or <i>getpmsg()</i> . |
| 17196 | [EINVAL] | An illegal value was specified by <i>flagsp</i> , or the STREAM or multiplexer
17197 referenced by <i>fildes</i> is linked (directly or indirectly) downstream from a
17198 multiplexer. |

17199 [ENOSTR] A STREAM is not associated with *fildev*.

17200 In addition, *getmsg()* and *getpmsg()* shall fail if the STREAM head had processed an
 17201 asynchronous error before the call. In this case, the value of *errno* does not reflect the result of
 17202 *getmsg()* or *getpmsg()* but reflects the prior error.

17203 EXAMPLES

17204 Getting Any Message

17205 In the following example, the value of *fd* is assumed to refer to an open STREAMS file. The call
 17206 to *getmsg()* retrieves any available message on the associated STREAM-head read queue,
 17207 returning control and data information to the buffers pointed to by *ctrlbuf* and *databuf*,
 17208 respectively.

```
17209 #include <stropts.h>
17210 ...
17211 int fd;
17212 char ctrlbuf[128];
17213 char databuf[512];
17214 struct strbuf ctrl;
17215 struct strbuf data;
17216 int flags = 0;
17217 int ret;

17218 ctrl.buf = ctrlbuf;
17219 ctrl.maxlen = sizeof(ctrlbuf);

17220 data.buf = databuf;
17221 data.maxlen = sizeof(databuf);

17222 ret = getmsg (fd, &ctrl, &data, &flags);
```

17223 Getting the First Message off the Queue

17224 In the following example, the call to *getpmsg()* retrieves the first available message on the
 17225 associated STREAM-head read queue.

```
17226 #include <stropts.h>
17227 ...

17228 int fd;
17229 char ctrlbuf[128];
17230 char databuf[512];
17231 struct strbuf ctrl;
17232 struct strbuf data;
17233 int band = 0;
17234 int flags = MSG_ANY;
17235 int ret;

17236 ctrl.buf = ctrlbuf;
17237 ctrl.maxlen = sizeof(ctrlbuf);

17238 data.buf = databuf;
17239 data.maxlen = sizeof(databuf);

17240 ret = getpmsg (fd, &ctrl, &data, &band, &flags);
```


17241 **APPLICATION USAGE**

17242 None.

17243 **RATIONALE**

17244 None.

17245 **FUTURE DIRECTIONS**

17246 None.

17247 **SEE ALSO**

17248 Section 2.6 (on page 38), *poll()*, *putmsg()*, *read()*, *write()*, the Base Definitions volume of
17249 IEEE Std 1003.1-2001, <**stropts.h**>

17250 **CHANGE HISTORY**

17251 First released in Issue 4, Version 2.

17252 **Issue 5**

17253 Moved from X/OPEN UNIX extension to BASE.

17254 A paragraph regarding “high-priority control parts of messages” is added to the RETURN
17255 VALUE section.

17256 **Issue 6**

17257 This function is marked as part of the XSI STREAMS Option Group.

17258 The **restrict** keyword is added to the *getmsg()* and *getpmsg()* prototypes for alignment with the
17259 ISO/IEC 9899:1999 standard.

17260 **NAME**

17261 getnameinfo — get name information

17262 **SYNOPSIS**

17263 #include <sys/socket.h>

17264 #include <netdb.h>

```
17265 int getnameinfo(const struct sockaddr *restrict sa, socklen_t salen,
17266 char *restrict node, socklen_t nodelen, char *restrict service,
17267 socklen_t servicelen, unsigned flags);
```

17268 **DESCRIPTION**

17269 The *getnameinfo()* function shall translate a socket address to a node name and service location,
 17270 all of which are defined as in *getaddrinfo()*.

17271 The *sa* argument points to a socket address structure to be translated.

17272 **IPv6** If the socket address structure contains an IPv4-mapped IPv6 address or an IPv4-compatible
 17273 IPv6 address, the implementation shall extract the embedded IPv4 address and lookup the node
 17274 name for that IPv4 address.

17275 **Note:** The IPv6 unspecified address ("::") and the IPv6 loopback address "::1") are not IPv4-
 17276 compatible addresses. If the address is the IPv6 unspecified address ("::"), a lookup is not
 17277 performed, and the [EAI_NONAME] error is returned.

17278 If the *node* argument is non-NULL and the *nodelen* argument is non-zero, then the *node* argument
 17279 points to a buffer able to contain up to *nodelen* characters that receives the node name as a null-
 17280 terminated string. If the *node* argument is NULL or the *nodelen* argument is zero, the node name
 17281 shall not be returned. If the node's name cannot be located, the numeric form of the node's
 17282 address is returned instead of its name.

17283 If the *service* argument is non-NULL and the *servicelen* argument is non-zero, then the *service*
 17284 argument points to a buffer able to contain up to *servicelen* bytes that receives the service name
 17285 as a null-terminated string. If the *service* argument is NULL or the *servicelen* argument is zero,
 17286 the service name shall not be returned. If the service's name cannot be located, the numeric form
 17287 of the service address (for example, its port number) shall be returned instead of its name.

17288 The *flags* argument is a flag that changes the default actions of the function. By default the fully-
 17289 qualified domain name (FQDN) for the host shall be returned, but:

- 17290 • If the flag bit NI_NOFQDN is set, only the node name portion of the FQDN shall be returned
 17291 for local hosts.
- 17292 • If the flag bit NI_NUMERICHOST is set, the numeric form of the host's address shall be
 17293 returned instead of its name, under all circumstances.
- 17294 • If the flag bit NI_NAMEREQD is set, an error shall be returned if the host's name cannot be
 17295 located.
- 17296 • If the flag bit NI_NUMERICSERV is set, the numeric form of the service address shall be
 17297 returned (for example, its port number) instead of its name, under all circumstances.
- 17298 • If the flag bit NI_DGRAM is set, this indicates that the service is a datagram service
 17299 (SOCK_DGRAM). The default behavior shall assume that the service is a stream service
 17300 (SOCK_STREAM).

17301 **Notes:**

- 17302 1. The two NI_NUMERICxxx flags are required to support the *-n* flag that many
 17303 commands provide.

17304 2. The NI_DGRAM flag is required for the few AF_INET and AF_INET6 port numbers (for
17305 example, [512,514]) that represent different services for UDP and TCP.

17306 The *getnameinfo()* function shall be thread-safe.

17307 RETURN VALUE

17308 A zero return value for *getnameinfo()* indicates successful completion; a non-zero return value
17309 indicates failure. The possible values for the failures are listed in the ERRORS section.

17310 Upon successful completion, *getnameinfo()* shall return the *node* and *service* names, if requested,
17311 in the buffers provided. The returned names are always null-terminated strings.

17312 ERRORS

17313 The *getnameinfo()* function shall fail and return the corresponding value if:

17314 [EAI_AGAIN] The name could not be resolved at this time. Future attempts may succeed.

17315 [EAI_BADFLAGS]

17316 The *flags* had an invalid value.

17317 [EAI_FAIL] A non-recoverable error occurred.

17318 [EAI_FAMILY] The address family was not recognized or the address length was invalid for
17319 the specified family.

17320 [EAI_MEMORY] There was a memory allocation failure.

17321 [EAI_NONAME] The name does not resolve for the supplied parameters.

17322 NI_NAMEREQD is set and the host's name cannot be located, or both
17323 *nodename* and *servname* were null.

17324 [EAI_SYSTEM] A system error occurred. The error code can be found in *errno*.

17325 EXAMPLES

17326 None.

17327 APPLICATION USAGE

17328 If the returned values are to be used as part of any further name resolution (for example, passed
17329 to *getaddrinfo()*), applications should provide buffers large enough to store any result possible
17330 on the system.

17331 Given the IPv4-mapped IPv6 address "::ffff:1.2.3.4", the implementation performs a
17332 lookup as if the socket address structure contains the IPv4 address "1.2.3.4".

17333 RATIONALE

17334 None.

17335 FUTURE DIRECTIONS

17336 None.

17337 SEE ALSO

17338 *gai_strerror()*, *getaddrinfo()*, *getservbyname()*, *inet_ntop()*, *socket()*, the Base Definitions volume of
17339 IEEE Std 1003.1-2001, <netdb.h>, <sys/socket.h>

17340 CHANGE HISTORY

17341 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

17342 The **restrict** keyword is added to the *getnameinfo()* prototype for alignment with the
17343 ISO/IEC 9899:1999 standard.

17344 NAME

17345 getnetbyaddr, getnetbyname, getnetent — network database functions

17346 SYNOPSIS

17347 #include <netdb.h>

17348 struct netent *getnetbyaddr(uint32_t *net*, int *type*);

17349 struct netent *getnetbyname(const char **name*);

17350 struct netent *getnetent(void);

17351 DESCRIPTION

17352 Refer to *endnetent()*.

17353 **NAME**

17354 getopt, optarg, opterr, optind, optopt — command option parsing

17355 **SYNOPSIS**

17356 #include <unistd.h>

17357 int getopt(int argc, char * const argv[], const char *optstring);

17358 extern char *optarg;

17359 extern int optind, opterr, optopt;

17360 **DESCRIPTION**

17361 The *getopt()* function is a command-line parser that shall follow Utility Syntax Guidelines 3, 4, 5,
 17362 6, 7, 9, and 10 in the Base Definitions volume of IEEE Std 1003.1-2001, Section 12.2, Utility Syntax
 17363 Guidelines.

17364 The parameters *argc* and *argv* are the argument count and argument array as passed to *main()*
 17365 (see *exec*). The argument *optstring* is a string of recognized option characters; if a character is
 17366 followed by a colon, the option takes an argument. All option characters allowed by Utility
 17367 Syntax Guideline 3 are allowed in *optstring*. The implementation may accept other characters as
 17368 an extension.

17369 The variable *optind* is the index of the next element of the *argv[]* vector to be processed. It shall
 17370 be initialized to 1 by the system, and *getopt()* shall update it when it finishes with each element
 17371 of *argv[]*. When an element of *argv[]* contains multiple option characters, it is unspecified how
 17372 *getopt()* determines which options have already been processed.

17373 The *getopt()* function shall return the next option character (if one is found) from *argv* that
 17374 matches a character in *optstring*, if there is one that matches. If the option takes an argument,
 17375 *getopt()* shall set the variable *optarg* to point to the option-argument as follows:

- 17376 1. If the option was the last character in the string pointed to by an element of *argv*, then
 17377 *optarg* shall contain the next element of *argv*, and *optind* shall be incremented by 2. If the
 17378 resulting value of *optind* is greater than *argc*, this indicates a missing option-argument, and
 17379 *getopt()* shall return an error indication.
- 17380 2. Otherwise, *optarg* shall point to the string following the option character in that element of
 17381 *argv*, and *optind* shall be incremented by 1.

17382 If, when *getopt()* is called:17383 *argv[optind]* is a null pointer17384 **argv[optind]* is not the character -17385 *argv[optind]* points to the string "--"17386 *getopt()* shall return -1 without changing *optind*. If:17387 *argv[optind]* points to the string "---"17388 *getopt()* shall return -1 after incrementing *optind*.

17389 If *getopt()* encounters an option character that is not contained in *optstring*, it shall return the
 17390 question-mark ('?') character. If it detects a missing option-argument, it shall return the colon
 17391 character (':') if the first character of *optstring* was a colon, or a question-mark character ('?')
 17392 otherwise. In either case, *getopt()* shall set the variable *optopt* to the option character that caused
 17393 the error. If the application has not set the variable *opterr* to 0 and the first character of *optstring*
 17394 is not a colon, *getopt()* shall also print a diagnostic message to *stderr* in the format specified for
 17395 the *getopts* utility.

17396 The *getopt()* function need not be reentrant. A function that is not required to be reentrant is not
 17397 required to be thread-safe.

17398 **RETURN VALUE**

17399 The *getopt()* function shall return the next option character specified on the command line.

17400 A colon (':') shall be returned if *getopt()* detects a missing argument and the first character of
17401 *optstring* was a colon (':').

17402 A question mark ('?') shall be returned if *getopt()* encounters an option character not in
17403 *optstring* or detects a missing argument and the first character of *optstring* was not a colon (':').

17404 Otherwise, *getopt()* shall return -1 when all command line options are parsed.

17405 **ERRORS**

17406 No errors are defined.

17407 **EXAMPLES**17408 **Parsing Command Line Options**

17409 The following code fragment shows how you might process the arguments for a utility that can
17410 take the mutually-exclusive options *a* and *b* and the options *f* and *o*, both of which require
17411 arguments:

```
17412 #include <unistd.h>
17413
17414 int
17415 main(int argc, char *argv[ ])
17416 {
17417     int c;
17418     int bflg, aflag, errflag;
17419     char *ifile;
17420     char *ofile;
17421     extern char *optarg;
17422     extern int optind, optopt;
17423     . . .
17424     while ((c = getopt(argc, argv, ":abf:o:")) != -1) {
17425         switch(c) {
17426             case 'a':
17427                 if (bflg)
17428                     errflag++;
17429                 else
17430                     aflag++;
17431                 break;
17432             case 'b':
17433                 if (aflag)
17434                     errflag++;
17435                 else {
17436                     bflg++;
17437                     bproc();
17438                 }
17439                 break;
17440             case 'f':
17441                 ifile = optarg;
17442                 break;
17443             case 'o':
17444                 ofile = optarg;
17445                 break;
```



```

17445         case '::':          /* -f or -o without operand */
17446             fprintf(stderr,
17447                 "Option -%c requires an operand\n", optopt);
17448             errflg++;
17449             break;
17450         case '?':
17451             fprintf(stderr,
17452                 "Unrecognized option: -%c\n", optopt);
17453             errflg++;
17454     }
17455 }
17456 if (errflg) {
17457     fprintf(stderr, "usage: . . . ");
17458     exit(2);
17459 }
17460 for ( ; optind < argc; optind++) {
17461     if (access(argv[optind], R_OK)) {
17462         . . .
17463     }

```

This code accepts any of the following as equivalent:

```

17465 cmd -ao arg path path
17466 cmd -a -o arg path path
17467 cmd -o arg -a path path
17468 cmd -a -o arg -- path path
17469 cmd -a -oarg path path
17470 cmd -aoarg path path

```

Checking Options and Arguments

The following example parses a set of command line options and prints messages to standard output for each option and argument that it encounters.

```

17474 #include <unistd.h>
17475 #include <stdio.h>
17476 ...
17477 int c;
17478 char *filename;
17479 extern char *optarg;
17480 extern int optind, optopt, opterr;
17481 ...
17482 while ((c = getopt(argc, argv, ":abf:")) != -1) {
17483     switch(c) {
17484         case 'a':
17485             printf("a is set\n");
17486             break;
17487         case 'b':
17488             printf("b is set\n");
17489             break;
17490         case 'f':
17491             filename = optarg;
17492             printf("filename is %s\n", filename);
17493             break;

```



```

17494         case '::':
17495             printf("-%c without filename\n", optopt);
17496             break;
17497         case '?':
17498             printf("unknown arg %c\n", optopt);
17499             break;
17500     }
17501 }

```

17502 Selecting Options from the Command Line

17503 The following example selects the type of database routines the user wants to use based on the
 17504 *Options* argument.

```

17505 #include <unistd.h>
17506 #include <string.h>
17507 ...
17508 char *Options = "hdbtl";
17509 ...
17510 int dbtype, i;
17511 char c;
17512 char *st;
17513 ...
17514 dbtype = 0;
17515 while ((c = getopt(argc, argv, Options)) != -1) {
17516     if ((st = strchr(Options, c)) != NULL) {
17517         dbtype = st - Options;
17518         break;
17519     }
17520 }

```

17521 APPLICATION USAGE

17522 The *getopt()* function is only required to support option characters included in Utility Syntax
 17523 Guideline 3. Many historical implementations of *getopt()* support other characters as options.
 17524 This is an allowed extension, but applications that use extensions are not maximally portable.
 17525 Note that support for multi-byte option characters is only possible when such characters can be
 17526 represented as type **int**.

17527 RATIONALE

17528 The *optopt* variable represents historical practice and allows the application to obtain the identity
 17529 of the invalid option.

17530 The description has been written to make it clear that *getopt()*, like the *getopts* utility, deals with
 17531 option-arguments whether separated from the option by <blank>s or not. Note that the
 17532 requirements on *getopt()* and *getopts* are more stringent than the Utility Syntax Guidelines.

17533 The *getopt()* function shall return **-1**, rather than EOF, so that <**stdio.h**> is not required.

17534 The special significance of a colon as the first character of *optstring* makes *getopt()* consistent
 17535 with the *getopts* utility. It allows an application to make a distinction between a missing
 17536 argument and an incorrect option letter without having to examine the option letter. It is true
 17537 that a missing argument can only be detected in one case, but that is a case that has to be
 17538 considered.

17539 **FUTURE DIRECTIONS**

17540 None.

17541 **SEE ALSO**

17542 *exec*, the Base Definitions volume of IEEE Std 1003.1-2001, <**unistd.h**>, the Shell and Utilities
17543 volume of IEEE Std 1003.1-2001

17544 **CHANGE HISTORY**

17545 First released in Issue 1. Derived from Issue 1 of the SVID.

17546 **Issue 5**17547 A note indicating that the *getopt()* function need not be reentrant is added to the DESCRIPTION.17548 **Issue 6**

17549 IEEE PASC Interpretation 1003.2 #150 is applied.

17550 **NAME**

17551 getpeername — get the name of the peer socket

17552 **SYNOPSIS**

17553 #include <sys/socket.h>

17554 int getpeername(int *socket*, struct sockaddr *restrict *address*,
17555 socklen_t *restrict *address_len*);

17556 **DESCRIPTION**

17557 The *getpeername()* function shall retrieve the peer address of the specified socket, store this
17558 address in the **sockaddr** structure pointed to by the *address* argument, and store the length of this
17559 address in the object pointed to by the *address_len* argument.

17560 If the actual length of the address is greater than the length of the supplied **sockaddr** structure,
17561 the stored address shall be truncated.

17562 If the protocol permits connections by unbound clients, and the peer is not bound, then the value
17563 stored in the object pointed to by *address* is unspecified.

17564 **RETURN VALUE**

17565 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
17566 indicate the error.

17567 **ERRORS**

17568 The *getpeername()* function shall fail if:

17569 [EBADF] The *socket* argument is not a valid file descriptor.

17570 [EINVAL] The socket has been shut down.

17571 [ENOTCONN] The socket is not connected or otherwise has not had the peer pre-specified.

17572 [ENOTSOCK] The *socket* argument does not refer to a socket.

17573 [EOPNOTSUPP] The operation is not supported for the socket protocol.

17574 The *getpeername()* function may fail if:

17575 [ENOBUFS] Insufficient resources were available in the system to complete the call.

17576 **EXAMPLES**

17577 None.

17578 **APPLICATION USAGE**

17579 None.

17580 **RATIONALE**

17581 None.

17582 **FUTURE DIRECTIONS**

17583 None.

17584 **SEE ALSO**

17585 *accept()*, *bind()*, *getsockname()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-2001,
17586 <sys/socket.h>

17587 **CHANGE HISTORY**

17588 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

17589 The **restrict** keyword is added to the *getpeername()* prototype for alignment with the
17590 ISO/IEC 9899:1999 standard.

17591 **NAME**

17592 getpgid — get the process group ID for a process

17593 **SYNOPSIS**

17594 XSI #include <unistd.h>

17595 pid_t getpgid(pid_t pid);

17596

17597 **DESCRIPTION**

17598 The *getpgid()* function shall return the process group ID of the process whose process ID is equal
 17599 to *pid*. If *pid* is equal to 0, *getpgid()* shall return the process group ID of the calling process.

17600 **RETURN VALUE**

17601 Upon successful completion, *getpgid()* shall return a process group ID. Otherwise, it shall return
 17602 (*pid_t*)-1 and set *errno* to indicate the error.

17603 **ERRORS**17604 The *getpgid()* function shall fail if:

17605 [EPERM] The process whose process ID is equal to *pid* is not in the same session as the
 17606 calling process, and the implementation does not allow access to the process
 17607 group ID of that process from the calling process.

17608 [ESRCH] There is no process with a process ID equal to *pid*.

17609 The *getpgid()* function may fail if:

17610 [EINVAL] The value of the *pid* argument is invalid.

17611 **EXAMPLES**

17612 None.

17613 **APPLICATION USAGE**

17614 None.

17615 **RATIONALE**

17616 None.

17617 **FUTURE DIRECTIONS**

17618 None.

17619 **SEE ALSO**

17620 *exec*, *fork()*, *getpgrp()*, *getpid()*, *getsid()*, *setpgid()*, *setsid()*, the Base Definitions volume of
 17621 IEEE Std 1003.1-2001, <unistd.h>

17622 **CHANGE HISTORY**

17623 First released in Issue 4, Version 2.

17624 **Issue 5**

17625 Moved from X/OPEN UNIX extension to BASE.

17626 **NAME**

17627 getpgrp — get the process group ID of the calling process

17628 **SYNOPSIS**

17629 #include <unistd.h>

17630 pid_t getpgrp(void);

17631 **DESCRIPTION**

17632 The *getpgrp()* function shall return the process group ID of the calling process.

17633 **RETURN VALUE**

17634 The *getpgrp()* function shall always be successful and no return value is reserved to indicate an
17635 error.

17636 **ERRORS**

17637 No errors are defined.

17638 **EXAMPLES**

17639 None.

17640 **APPLICATION USAGE**

17641 None.

17642 **RATIONALE**

17643 4.3 BSD provides a *getpgrp()* function that returns the process group ID for a specified process.
17644 Although this function supports job control, all known job control shells always specify the
17645 calling process with this function. Thus, the simpler System V *getpgrp()* suffices, and the added
17646 complexity of the 4.3 BSD *getpgrp()* is provided by the XSI extension *getpgid()*.

17647 **FUTURE DIRECTIONS**

17648 None.

17649 **SEE ALSO**

17650 *exec*, *fork()*, *getpgid()*, *getpid()*, *getppid()*, *kill()*, *setpgid()*, *setsid()*, the Base Definitions volume of
17651 IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

17652 **CHANGE HISTORY**

17653 First released in Issue 1. Derived from Issue 1 of the SVID.

17654 **Issue 6**

17655 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

17656 The following new requirements on POSIX implementations derive from alignment with the
17657 Single UNIX Specification:

- 17658
 - The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
17659 required for conforming implementations of previous POSIX specifications, it was not
17660 required for UNIX applications.

17661 **NAME**

17662 getpid — get the process ID

17663 **SYNOPSIS**

17664 #include <unistd.h>

17665 pid_t getpid(void);

17666 **DESCRIPTION**17667 The *getpid()* function shall return the process ID of the calling process.17668 **RETURN VALUE**17669 The *getpid()* function shall always be successful and no return value is reserved to indicate an error.17671 **ERRORS**

17672 No errors are defined.

17673 **EXAMPLES**

17674 None.

17675 **APPLICATION USAGE**

17676 None.

17677 **RATIONALE**

17678 None.

17679 **FUTURE DIRECTIONS**

17680 None.

17681 **SEE ALSO**17682 *exec*, *fork()*, *getpgrp()*, *getppid()*, *kill()*, *setpgid()*, *setsid()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>17684 **CHANGE HISTORY**

17685 First released in Issue 1. Derived from Issue 1 of the SVID.

17686 **Issue 6**

17687 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

17688 The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- 17690
- The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

17693 NAME

17694 getpmsg — receive next message from a STREAMS file

17695 SYNOPSIS

17696 XSI #include <stropts.h>

```
17697       int getpmsg(int fildes, struct strbuf *restrict ctlptr,  
17698                   struct strbuf *restrict dataptr, int *restrict bandp,  
17699                   int *restrict flagsp);
```

17700

17701 DESCRIPTION

17702 Refer to *getmsg()*.

17703 **NAME**

17704 getppid — get the parent process ID

17705 **SYNOPSIS**

17706 #include <unistd.h>

17707 pid_t getppid(void);

17708 **DESCRIPTION**

17709 The *getppid()* function shall return the parent process ID of the calling process.

17710 **RETURN VALUE**

17711 The *getppid()* function shall always be successful and no return value is reserved to indicate an
17712 error.

17713 **ERRORS**

17714 No errors are defined.

17715 **EXAMPLES**

17716 None.

17717 **APPLICATION USAGE**

17718 None.

17719 **RATIONALE**

17720 None.

17721 **FUTURE DIRECTIONS**

17722 None.

17723 **SEE ALSO**

17724 *exec*, *fork()*, *getpgid()*, *getpgrp()*, *getpid()*, *kill()*, *setpgid()*, *setsid()*, the Base Definitions volume of
17725 IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

17726 **CHANGE HISTORY**

17727 First released in Issue 1. Derived from Issue 1 of the SVID.

17728 **Issue 6**

17729 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

17730 The following new requirements on POSIX implementations derive from alignment with the
17731 Single UNIX Specification:

- 17732
 - The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
17733 required for conforming implementations of previous POSIX specifications, it was not
17734 required for UNIX applications.

17735 **NAME**

17736 getpriority, setpriority — get and set the nice value

17737 **SYNOPSIS**

17738 XSI #include <sys/resource.h>

17739 int getpriority(int which, id_t who);

17740 int setpriority(int which, id_t who, int value);

17741

17742 **DESCRIPTION**

17743 The *getpriority()* function shall obtain the nice value of a process, process group, or user. The
 17744 *setpriority()* function shall set the nice value of a process, process group, or user to
 17745 value+{NZERO}.

17746 Target processes are specified by the values of the *which* and *who* arguments. The *which*
 17747 argument may be one of the following values: PRIO_PROCESS, PRIO_PGRP, or PRIO_USER,
 17748 indicating that the *who* argument is to be interpreted as a process ID, a process group ID, or an
 17749 effective user ID, respectively. A 0 value for the *who* argument specifies the current process,
 17750 process group, or user.

17751 The nice value set with *setpriority()* shall be applied to the process. If the process is multi-
 17752 threaded, the nice value shall affect all system scope threads in the process.

17753 If more than one process is specified, *getpriority()* shall return value {NZERO} less than the
 17754 lowest nice value pertaining to any of the specified processes, and *setpriority()* shall set the nice
 17755 values of all of the specified processes to value+{NZERO}.

17756 The default nice value is {NZERO}; lower nice values shall cause more favorable scheduling.
 17757 While the range of valid nice values is [0,{NZERO}*2-1], implementations may enforce more
 17758 restrictive limits. If value+{NZERO} is less than the system's lowest supported nice value,
 17759 *setpriority()* shall set the nice value to the lowest supported value; if value+{NZERO} is greater
 17760 than the system's highest supported nice value, *setpriority()* shall set the nice value to the highest
 17761 supported value.

17762 Only a process with appropriate privileges can lower its nice value.

17763 PS|TPS Any processes or threads using SCHED_FIFO or SCHED_RR shall be unaffected by a call to
 17764 *setpriority()*. This is not considered an error. A process which subsequently reverts to
 17765 SCHED_OTHER need not have its priority affected by such a *setpriority()* call.

17766 The effect of changing the nice value may vary depending on the process-scheduling algorithm
 17767 in effect.

17768 Since *getpriority()* can return the value -1 on successful completion, it is necessary to set *errno* to
 17769 0 prior to a call to *getpriority()*. If *getpriority()* returns the value -1, then *errno* can be checked to
 17770 see if an error occurred or if the value is a legitimate nice value.

17771 **RETURN VALUE**

17772 Upon successful completion, *getpriority()* shall return an integer in the range -{NZERO} to
 17773 {NZERO}-1. Otherwise, -1 shall be returned and *errno* set to indicate the error.

17774 Upon successful completion, *setpriority()* shall return 0; otherwise, -1 shall be returned and *errno*
 17775 set to indicate the error.

17776 **ERRORS**

17777 The *getpriority()* and *setpriority()* functions shall fail if:

17778 [ESRCH] No process could be located using the *which* and *who* argument values
 17779 specified.

17780 [EINVAL] The value of the *which* argument was not recognized, or the value of the *who*
 17781 argument is not a valid process ID, process group ID, or user ID.

17782 In addition, *setpriority()* may fail if:

17783 [EPERM] A process was located, but neither the real nor effective user ID of the
 17784 executing process match the effective user ID of the process whose nice value
 17785 is being changed.

17786 [EACCES] A request was made to change the nice value to a lower numeric value and
 17787 the current process does not have appropriate privileges.

17788 EXAMPLES

17789 Using *getpriority()*

17790 The following example returns the current scheduling priority for the process ID returned by the
 17791 call to *getpid()*.

```
17792 #include <sys/resource.h>
17793 ...
17794 int which = PRIO_PROCESS;
17795 id_t pid;
17796 int ret;

17797 pid = getpid();
17798 ret = getpriority(which, pid);
```

17799 Using *setpriority()*

17800 The following example sets the priority for the current process ID to -20.

```
17801 #include <sys/resource.h>
17802 ...
17803 int which = PRIO_PROCESS;
17804 id_t pid;
17805 int priority = -20;
17806 int ret;

17807 pid = getpid();
17808 ret = setpriority(which, pid, priority);
```

17809 APPLICATION USAGE

17810 The *getpriority()* and *setpriority()* functions work with an offset nice value (nice value
 17811 -{NZERO}). The nice value is in the range [0,2*{NZERO} -1], while the return value for
 17812 *getpriority()* and the third parameter for *setpriority()* are in the range [-{NZERO},{NZERO} -1].

17813 RATIONALE

17814 None.

17815 FUTURE DIRECTIONS

17816 None.

17817 SEE ALSO

17818 *nice()*, *sched_get_priority_max()*, *sched_setscheduler()*, the Base Definitions volume of
 17819 IEEE Std 1003.1-2001, <sys/resource.h>

17820 **CHANGE HISTORY**

17821 First released in Issue 4, Version 2.

17822 **Issue 5**

17823 Moved from X/OPEN UNIX extension to BASE.

17824 The DESCRIPTION is reworded in terms of the nice value rather than *priority* to avoid confusion
17825 with functionality in the POSIX Realtime Extension.

17826 **NAME**

17827 getprotobyname, getprotobynumber, getprotent — network protocol database functions

17828 **SYNOPSIS**

17829 #include <netdb.h>

17830 struct protoent *getprotobyname(const char *name);

17831 struct protoent *getprotobynumber(int proto);

17832 struct protoent *getprotoent(void);

17833 **DESCRIPTION**

17834 Refer to *endprotoent()*.

17835 NAME

17836 getpwent — get user database entry

17837 SYNOPSIS

17838 xSI #include <pwd.h>

17839 struct passwd *getpwent(void);

17840

17841 DESCRIPTION

17842 Refer to *endpwent()*.

17843 NAME

17844 getpwnam, getpwnam_r — search user database for a name

17845 SYNOPSIS

17846 #include <pwd.h>

17847 struct passwd *getpwnam(const char *name);

17848 TSF int getpwnam_r(const char *name, struct passwd *pwd, char *buffer,

17849 size_t bufsize, struct passwd **result);

17850

17851 DESCRIPTION

17852 The *getpwnam()* function shall search the user database for an entry with a matching *name*.17853 The *getpwnam()* function need not be reentrant. A function that is not required to be reentrant is
17854 not required to be thread-safe.17855 Applications wishing to check for error situations should set *errno* to 0 before calling
17856 *getpwnam()*. If *getpwnam()* returns a null pointer and *errno* is non-zero, an error occurred.17857 TSF The *getpwnam_r()* function shall update the **passwd** structure pointed to by *pwd* and store a
17858 pointer to that structure at the location pointed to by *result*. The structure shall contain an entry
17859 from the user database with a matching *name*. Storage referenced by the structure is allocated
17860 from the memory provided with the *buffer* parameter, which is *bufsize* bytes in size. The
17861 maximum size needed for this buffer can be determined with the {_SC_GETPW_R_SIZE_MAX}
17862 *sysconf()* parameter. A NULL pointer shall be returned at the location pointed to by *result* on
17863 error or if the requested entry is not found.

17864 RETURN VALUE

17865 The *getpwnam()* function shall return a pointer to a **struct passwd** with the structure as defined
17866 in <pwd.h> with a matching entry if found. A null pointer shall be returned if the requested
17867 entry is not found, or an error occurs. On error, *errno* shall be set to indicate the error.17868 The return value may point to a static area which is overwritten by a subsequent call to
17869 *getpwent()*, *getpwnam()*, or *getpwuid()*.17870 TSF If successful, the *getpwnam_r()* function shall return zero; otherwise, an error number shall be
17871 returned to indicate the error.

17872 ERRORS

17873 The *getpwnam()* and *getpwnam_r()* functions may fail if:

17874 [EIO] An I/O error has occurred.

17875 [EINTR] A signal was caught during *getpwnam()*.

17876 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

17877 [ENFILE] The maximum allowable number of files is currently open in the system.

17878 The *getpwnam_r()* function may fail if:17879 TSF [ERANGE] Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to
17880 be referenced by the resulting **passwd** structure.

17881 **EXAMPLES**17882 **Getting an Entry for the Login Name**

17883 The following example uses the *getlogin()* function to return the name of the user who logged in;
 17884 this information is passed to the *getpwnam()* function to get the user database entry for that user.

```

17885 #include <sys/types.h>
17886 #include <pwd.h>
17887 #include <unistd.h>
17888 #include <stdio.h>
17889 #include <stdlib.h>
17890 ...
17891 char *lgn;
17892 struct passwd *pw;
17893 ...
17894 if ((lgn = getlogin()) == NULL || (pw = getpwnam(lgn)) == NULL) {
17895     fprintf(stderr, "Get of user information failed.\n"); exit(1);
17896 }
17897 ...

```

17898 **APPLICATION USAGE**

17899 Three names associated with the current process can be determined: *getpwuid(geteuid())* returns
 17900 the name associated with the effective user ID of the process; *getlogin()* returns the name
 17901 associated with the current login activity; and *getpwuid(getuid())* returns the name associated
 17902 with the real user ID of the process.

17903 The *getpwnam_r()* function is thread-safe and returns values in a user-supplied buffer instead of
 17904 possibly using a static data area that may be overwritten by each call.

17905 **RATIONALE**

17906 None.

17907 **FUTURE DIRECTIONS**

17908 None.

17909 **SEE ALSO**

17910 *getpwuid()*, the Base Definitions volume of IEEE Std 1003.1-2001, *<limits.h>*, *<pwd.h>*,
 17911 *<sys/types.h>*

17912 **CHANGE HISTORY**

17913 First released in Issue 1. Derived from System V Release 2.0.

17914 **Issue 5**

17915 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
 17916 VALUE section.

17917 The *getpwnam_r()* function is included for alignment with the POSIX Threads Extension.

17918 A note indicating that the *getpwnam()* function need not be reentrant is added to the
 17919 DESCRIPTION.

17920 **Issue 6**

17921 The *getpwnam_r()* function is marked as part of the Thread-Safe Functions option.

17922 The Open Group Corrigendum U028/3 is applied, correcting text in the DESCRIPTION
 17923 describing matching the *name*.

- 17924 In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.
- 17925 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.
- 17926 The following new requirements on POSIX implementations derive from alignment with the
17927 Single UNIX Specification:
- 17928 • The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was
17929 required for conforming implementations of previous POSIX specifications, it was not
17930 required for UNIX applications.
 - 17931 • In the RETURN VALUE section, the requirement to set *errno* on error is added.
 - 17932 • The [EMFILE], [ENFILE], and [ENXIO] optional error conditions are added.
- 17933 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
17934 its avoidance of possibly using a static data area.
- 17935 IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the
17936 buffer from *bufsize* characters to bytes.

17937 NAME

17938 getpwuid, getpwuid_r — search user database for a user ID

17939 SYNOPSIS

17940 #include <pwd.h>

17941 struct passwd *getpwuid(uid_t uid);

17942 TSF int getpwuid_r(uid_t uid, struct passwd *pwd, char *buffer,

17943 size_t bufsize, struct passwd **result);

17944

17945 DESCRIPTION

17946 The *getpwuid()* function shall search the user database for an entry with a matching *uid*.17947 The *getpwuid()* function need not be reentrant. A function that is not required to be reentrant is
17948 not required to be thread-safe.17949 Applications wishing to check for error situations should set *errno* to 0 before calling *getpwuid()*.17950 If *getpwuid()* returns a null pointer and *errno* is set to non-zero, an error occurred.17951 TSF The *getpwuid_r()* function shall update the **passwd** structure pointed to by *pwd* and store a
17952 pointer to that structure at the location pointed to by *result*. The structure shall contain an entry
17953 from the user database with a matching *uid*. Storage referenced by the structure is allocated
17954 from the memory provided with the *buffer* parameter, which is *bufsize* bytes in size. The
17955 maximum size needed for this buffer can be determined with the {_SC_GETPW_R_SIZE_MAX}
17956 *sysconf()* parameter. A NULL pointer shall be returned at the location pointed to by *result* on
17957 error or if the requested entry is not found.

17958 RETURN VALUE

17959 The *getpwuid()* function shall return a pointer to a **struct passwd** with the structure as defined in
17960 <**pwd.h**> with a matching entry if found. A null pointer shall be returned if the requested entry
17961 is not found, or an error occurs. On error, *errno* shall be set to indicate the error.17962 The return value may point to a static area which is overwritten by a subsequent call to
17963 *getpwent()*, *getpwnam()*, or *getpwuid()*.17964 TSF If successful, the *getpwuid_r()* function shall return zero; otherwise, an error number shall be
17965 returned to indicate the error.

17966 ERRORS

17967 The *getpwuid()* and *getpwuid_r()* functions may fail if:

17968 [EIO] An I/O error has occurred.

17969 [EINTR] A signal was caught during *getpwuid()*.

17970 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

17971 [ENFILE] The maximum allowable number of files is currently open in the system.

17972 The *getpwuid_r()* function may fail if:17973 TSF [ERANGE] Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to
17974 be referenced by the resulting **passwd** structure.

17975 **EXAMPLES**17976 **Getting an Entry for the Root User**

17977 The following example gets the user database entry for the user with user ID 0 (root).

```
17978 #include <sys/types.h>
17979 #include <pwd.h>
17980 ...
17981 uid_t id = 0;
17982 struct passwd *pwd;
17983 pwd = getpwuid(id);
```

17984 **Finding the Name for the Effective User ID**

17985 The following example defines *pws* as a pointer to a structure of type **passwd**, which is used to
 17986 store the structure pointer returned by the call to the *getpwuid()* function. The *geteuid()* function
 17987 shall return the effective user ID of the calling process; this is used as the search criteria for the
 17988 *getpwuid()* function. The call to *getpwuid()* shall return a pointer to the structure containing that
 17989 user ID value.

```
17990 #include <unistd.h>
17991 #include <sys/types.h>
17992 #include <pwd.h>
17993 ...
17994 struct passwd *pws;
17995 pws = getpwuid(geteuid());
```

17996 **Finding an Entry in the User Database**

17997 The following example uses *getpwuid()* to search the user database for a user ID that was
 17998 previously stored in a **stat** structure, then prints out the user name if it is found. If the user is not
 17999 found, the program prints the numeric value of the user ID for the entry.

```
18000 #include <sys/types.h>
18001 #include <pwd.h>
18002 #include <stdio.h>
18003 ...
18004 struct stat statbuf;
18005 struct passwd *pwd;
18006 ...
18007 if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
18008     printf(" %-8.8s", pwd->pw_name);
18009 else
18010     printf(" %-8d", statbuf.st_uid);
```

18011 **APPLICATION USAGE**

18012 Three names associated with the current process can be determined: *getpwuid(geteuid())* returns
 18013 the name associated with the effective user ID of the process; *getlogin()* returns the name
 18014 associated with the current login activity; and *getpwuid(getuid())* returns the name associated
 18015 with the real user ID of the process.

18016 The *getpwuid_r()* function is thread-safe and returns values in a user-supplied buffer instead of
 18017 possibly using a static data area that may be overwritten by each call.

18018 **RATIONALE**

18019 None.

18020 **FUTURE DIRECTIONS**

18021 None.

18022 **SEE ALSO**

18023 *getpwnam()*, *geteuid()*, *getuid()*, *getlogin()*, the Base Definitions volume of IEEE Std 1003.1-2001,
 18024 *<limits.h>*, *<pwd.h>*, *<sys/types.h>*

18025 **CHANGE HISTORY**

18026 First released in Issue 1. Derived from System V Release 2.0.

18027 **Issue 5**

18028 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
 18029 VALUE section.

18030 The *getpwuid_r()* function is included for alignment with the POSIX Threads Extension.

18031 A note indicating that the *getpwuid()* function need not be reentrant is added to the
 18032 DESCRIPTION.

18033 **Issue 6**18034 The *getpwuid_r()* function is marked as part of the Thread-Safe Functions option.

18035 The Open Group Corrigendum U028/3 is applied, correcting text in the DESCRIPTION
 18036 describing matching the *uid*.

18037 In the SYNOPSIS, the optional include of the *<sys/types.h>* header is removed.

18038 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

18039 The following new requirements on POSIX implementations derive from alignment with the
 18040 Single UNIX Specification:

18041 • The requirement to include *<sys/types.h>* has been removed. Although *<sys/types.h>* was
 18042 required for conforming implementations of previous POSIX specifications, it was not
 18043 required for UNIX applications.

18044 • In the RETURN VALUE section, the requirement to set *errno* on error is added.

18045 • The [EIO], [EINTR], [EMFILE], and [ENFILE] optional error conditions are added.

18046 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
 18047 its avoidance of possibly using a static data area.

18048 IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the
 18049 buffer from *bufsize* characters to bytes.

18050 NAME

18051 getrlimit, setrlimit — control maximum resource consumption

18052 SYNOPSIS

18053 XSI

```
#include <sys/resource.h>
```

18054

```
int getrlimit(int resource, struct rlimit *rlp);
```

18055

```
int setrlimit(int resource, const struct rlimit *rlp);
```

18056

18057 DESCRIPTION

18058 The *getrlimit()* function shall get, and the *setrlimit()* function shall set, limits on the consumption
18059 of a variety of resources.

18060 Each call to either *getrlimit()* or *setrlimit()* identifies a specific resource to be operated upon as
 18061 well as a resource limit. A resource limit is represented by an **rlimit** structure. The *rlim_cur*
 18062 member specifies the current or soft limit and the *rlim_max* member specifies the maximum or
 18063 hard limit. Soft limits may be changed by a process to any value that is less than or equal to the
 18064 hard limit. A process may (irreversibly) lower its hard limit to any value that is greater than or
 18065 equal to the soft limit. Only a process with appropriate privileges can raise a hard limit. Both
 18066 hard and soft limits can be changed in a single call to *setrlimit()* subject to the constraints
 18067 described above.

18068 The value RLIM_INFINITY, defined in *<sys/resource.h>*, shall be considered to be larger than
 18069 any other limit value. If a call to *getrlimit()* returns RLIM_INFINITY for a resource, it means the
 18070 implementation shall not enforce limits on that resource. Specifying RLIM_INFINITY as any
 18071 resource limit value on a successful call to *setrlimit()* shall inhibit enforcement of that resource
 18072 limit.

18073 The following resources are defined:

18074 RLIMIT_CORE This is the maximum size of a **core** file, in bytes, that may be created by a
 18075 process. A limit of 0 shall prevent the creation of a **core** file. If this limit is
 18076 exceeded, the writing of a **core** file shall terminate at this size.

18077 RLIMIT_CPU This is the maximum amount of CPU time, in seconds, used by a process.
 18078 If this limit is exceeded, SIGXCPU shall be generated for the process. If
 18079 the process is catching or ignoring SIGXCPU, or all threads belonging to
 18080 that process are blocking SIGXCPU, the behavior is unspecified.

18081 RLIMIT_DATA This is the maximum size of a process' data segment, in bytes. If this limit
 18082 is exceeded, the *malloc()* function shall fail with *errno* set to [ENOMEM].

18083 RLIMIT_FSIZE This is the maximum size of a file, in bytes, that may be created by a
 18084 process. If a write or truncate operation would cause this limit to be
 18085 exceeded, SIGXFSZ shall be generated for the thread. If the thread is
 18086 blocking, or the process is catching or ignoring SIGXFSZ, continued
 18087 attempts to increase the size of a file from end-of-file to beyond the limit
 18088 shall fail with *errno* set to [EFBIG].

18089 RLIMIT_NOFILE This is a number one greater than the maximum value that the system
 18090 may assign to a newly-created descriptor. If this limit is exceeded,
 18091 functions that allocate new file descriptors may fail with *errno* set to
 18092 [EMFILE]. This limit constrains the number of file descriptors that a
 18093 process may allocate.

18094 RLIMIT_STACK This is the maximum size of a process' stack, in bytes. The
 18095 implementation does not automatically grow the stack beyond this limit.

18096 If this limit is exceeded, SIGSEGV shall be generated for the thread. If the
 18097 thread is blocking SIGSEGV, or the process is ignoring or catching
 18098 SIGSEGV and has not made arrangements to use an alternate stack, the
 18099 disposition of SIGSEGV shall be set to SIG_DFL before it is generated.

18100 RLIMIT_AS This is the maximum size of a process' total available memory, in bytes. If
 18101 this limit is exceeded, the *malloc()* and *mmap()* functions shall fail with
 18102 *errno* set to [ENOMEM]. In addition, the automatic stack growth fails
 18103 with the effects outlined above.

18104 When using the *getrlimit()* function, if a resource limit can be represented correctly in an object
 18105 of type **rlim_t**, then its representation is returned; otherwise, if the value of the resource limit is
 18106 equal to that of the corresponding saved hard limit, the value returned shall be
 18107 RLIM_SAVED_MAX; otherwise, the value returned shall be RLIM_SAVED_CUR.

18108 When using the *setrlimit()* function, if the requested new limit is RLIM_INFINITY, the new limit
 18109 shall be “no limit”; otherwise, if the requested new limit is RLIM_SAVED_MAX, the new limit
 18110 shall be the corresponding saved hard limit; otherwise, if the requested new limit is
 18111 RLIM_SAVED_CUR, the new limit shall be the corresponding saved soft limit; otherwise, the
 18112 new limit shall be the requested value. In addition, if the corresponding saved limit can be
 18113 represented correctly in an object of type **rlim_t** then it shall be overwritten with the new limit.

18114 The result of setting a limit to RLIM_SAVED_MAX or RLIM_SAVED_CUR is unspecified unless
 18115 a previous call to *getrlimit()* returned that value as the soft or hard limit for the corresponding
 18116 resource limit.

18117 The determination of whether a limit can be correctly represented in an object of type **rlim_t** is
 18118 implementation-defined. For example, some implementations permit a limit whose value is
 18119 greater than RLIM_INFINITY and others do not.

18120 The *exec* family of functions shall cause resource limits to be saved.

18121 **RETURN VALUE**

18122 Upon successful completion, *getrlimit()* and *setrlimit()* shall return 0. Otherwise, these functions
 18123 shall return -1 and set *errno* to indicate the error.

18124 **ERRORS**

18125 The *getrlimit()* and *setrlimit()* functions shall fail if:

18126 [EINVAL] An invalid *resource* was specified; or in a *setrlimit()* call, the new *rlim_cur*
 18127 exceeds the new *rlim_max*.

18128 [EPERM] The limit specified to *setrlimit()* would have raised the maximum limit value,
 18129 and the calling process does not have appropriate privileges.

18130 The *setrlimit()* function may fail if:

18131 [EINVAL] The limit specified cannot be lowered because current usage is already higher
 18132 than the limit.

18133 EXAMPLES

18134 None.

18135 APPLICATION USAGE

18136 If a process attempts to set the hard limit or soft limit for RLIMIT_NOFILE to less than the value
18137 of {_POSIX_OPEN_MAX} from <limits.h>, unexpected behavior may occur.

18138 RATIONALE

18139 None.

18140 FUTURE DIRECTIONS

18141 None.

18142 SEE ALSO

18143 *exec*, *fork()*, *malloc()*, *open()*, *sigaltstack()*, *sysconf()*, *ulimit()*, the Base Definitions volume of
18144 IEEE Std 1003.1-2001, <stropts.h>, <sys/resource.h>

18145 CHANGE HISTORY

18146 First released in Issue 4, Version 2.

18147 Issue 5

18148 Moved from X/OPEN UNIX extension to BASE.

18149 An APPLICATION USAGE section is added.

18150 Large File Summit extensions are added.

18151 **NAME**18152 `getrusage` — get information about resource utilization18153 **SYNOPSIS**18154 XSI `#include <sys/resource.h>`18155 `int getrusage(int who, struct rusage *r_usage);`

18156

18157 **DESCRIPTION**

18158 The `getrusage()` function shall provide measures of the resources used by the current process or
 18159 its terminated and waited-for child processes. If the value of the *who* argument is
 18160 `RUSAGE_SELF`, information shall be returned about resources used by the current process. If the
 18161 value of the *who* argument is `RUSAGE_CHILDREN`, information shall be returned about
 18162 resources used by the terminated and waited-for children of the current process. If the child is
 18163 never waited for (for example, if the parent has `SA_NOCLDWAIT` set or sets `SIGCHLD` to
 18164 `SIG_IGN`), the resource information for the child process is discarded and not included in the
 18165 resource information provided by `getrusage()`.

18166 The *r_usage* argument is a pointer to an object of type **struct rusage** in which the returned
 18167 information is stored.

18168 **RETURN VALUE**

18169 Upon successful completion, `getrusage()` shall return 0; otherwise, `-1` shall be returned and *errno*
 18170 set to indicate the error.

18171 **ERRORS**18172 The `getrusage()` function shall fail if:18173 `[EINVAL]` The value of the *who* argument is not valid.18174 **EXAMPLES**18175 **Using `getrusage()`**

18176 The following example returns information about the resources used by the current process.

```
18177        #include <sys/resource.h>
18178        ...
18179        int who = RUSAGE_SELF;
18180        struct rusage usage;
18181        int ret;

18182        ret = getrusage(who, &usage);
```

18183 **APPLICATION USAGE**

18184 None.

18185 **RATIONALE**

18186 None.

18187 **FUTURE DIRECTIONS**

18188 None.

18189 **SEE ALSO**

18190 `exit()`, `sigaction()`, `time()`, `times()`, `wait()`, the Base Definitions volume of IEEE Std 1003.1-2001,
 18191 `<sys/resource.h>`

18192 **CHANGE HISTORY**

18193 First released in Issue 4, Version 2.

18194 **Issue 5**

18195 Moved from X/OPEN UNIX extension to BASE.

18196 **NAME**

18197 gets — get a string from a stdin stream

18198 **SYNOPSIS**

18199 #include <stdio.h>

18200 char *gets(char *s);

18201 **DESCRIPTION**

18202 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 18203 conflict between the requirements described here and the ISO C standard is unintentional. This
 18204 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

18205 The *gets()* function shall read bytes from the standard input stream, *stdin*, into the array pointed
 18206 to by *s*, until a <newline> is read or an end-of-file condition is encountered. Any <newline> shall
 18207 be discarded and a null byte shall be placed immediately after the last byte read into the array.

18208 CX The *gets()* function may mark the *st_atime* field of the file associated with *stream* for update. The
 18209 *st_atime* field shall be marked for update by the first successful execution of *fgetc()*, *fgets()*,
 18210 *fread()*, *getc()*, *getchar()*, *gets()*, *fscanf()*, or *scanf()* using *stream* that returns data not supplied by
 18211 a prior call to *ungetc()*.

18212 **RETURN VALUE**

18213 Upon successful completion, *gets()* shall return *s*. If the stream is at end-of-file, the end-of-file
 18214 indicator for the stream shall be set and *gets()* shall return a null pointer. If a read error occurs,
 18215 CX the error indicator for the stream shall be set, *gets()* shall return a null pointer, and set *errno* to
 18216 indicate the error.

18217 **ERRORS**18218 Refer to *fgetc()*.18219 **EXAMPLES**

18220 None.

18221 **APPLICATION USAGE**

18222 Reading a line that overflows the array pointed to by *s* results in undefined behavior. The use of
 18223 *fgets()* is recommended.

18224 Since the user cannot specify the length of the buffer passed to *gets()*, use of this function is
 18225 discouraged. The length of the string read is unlimited. It is possible to overflow this buffer in
 18226 such a way as to cause applications to fail, or possible system security violations.

18227 It is recommended that the *fgets()* function should be used to read input lines.

18228 **RATIONALE**

18229 None.

18230 **FUTURE DIRECTIONS**

18231 None.

18232 **SEE ALSO**18233 *feof()*, *ferror()*, *fgets()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>18234 **CHANGE HISTORY**

18235 First released in Issue 1. Derived from Issue 1 of the SVID.

18236 **Issue 6**

18237 Extensions beyond the ISO C standard are marked.

18238 **NAME**

18239 getservbyname, getservbyport, getservent — network services database functions

18240 **SYNOPSIS**

18241 #include <netdb.h>

18242 struct servent *getservbyname(const char *name, const char *proto);

18243 struct servent *getservbyport(int port, const char *proto);

18244 struct servent *getservent(void);

18245 **DESCRIPTION**

18246 Refer to *endservent()*.

18247 NAME

18248 getsid — get the process group ID of a session leader

18249 SYNOPSIS

18250 XSI `#include <unistd.h>`

18251 `pid_t getsid(pid_t pid);`

18252

18253 DESCRIPTION

18254 The *getsid()* function shall obtain the process group ID of the process that is the session leader of
18255 the process specified by *pid*. If *pid* is (**pid_t**)0, it specifies the calling process.

18256 RETURN VALUE

18257 Upon successful completion, *getsid()* shall return the process group ID of the session leader of
18258 the specified process. Otherwise, it shall return (**pid_t**)−1 and set *errno* to indicate the error.

18259 ERRORS

18260 The *getsid()* function shall fail if:

18261 [EPERM] The process specified by *pid* is not in the same session as the calling process,
18262 and the implementation does not allow access to the process group ID of the
18263 session leader of that process from the calling process.

18264 [ESRCH] There is no process with a process ID equal to *pid*.

18265 EXAMPLES

18266 None.

18267 APPLICATION USAGE

18268 None.

18269 RATIONALE

18270 None.

18271 FUTURE DIRECTIONS

18272 None.

18273 SEE ALSO

18274 *exec*, *fork()*, *getpid()*, *getpgid()*, *setpgid()*, *setsid()*, the Base Definitions volume of
18275 IEEE Std 1003.1-2001, <**unistd.h**>

18276 CHANGE HISTORY

18277 First released in Issue 4, Version 2.

18278 Issue 5

18279 Moved from X/OPEN UNIX extension to BASE.

18280 **NAME**

18281 getsockname — get the socket name

18282 **SYNOPSIS**

18283 #include <sys/socket.h>

```
18284       int getsockname(int socket, struct sockaddr *restrict address,  
18285                       socklen_t *restrict address_len);
```

18286 **DESCRIPTION**

18287 The *getsockname()* function shall retrieve the locally-bound name of the specified socket, store
18288 this address in the **sockaddr** structure pointed to by the *address* argument, and store the length of
18289 this address in the object pointed to by the *address_len* argument.

18290 If the actual length of the address is greater than the length of the supplied **sockaddr** structure,
18291 the stored address shall be truncated.

18292 If the socket has not been bound to a local name, the value stored in the object pointed to by
18293 *address* is unspecified.

18294 **RETURN VALUE**

18295 Upon successful completion, 0 shall be returned, the *address* argument shall point to the address
18296 of the socket, and the *address_len* argument shall point to the length of the address. Otherwise, -1
18297 shall be returned and *errno* set to indicate the error.

18298 **ERRORS**18299 The *getsockname()* function shall fail if:18300 [EBADF] The *socket* argument is not a valid file descriptor.18301 [ENOTSOCK] The *socket* argument does not refer to a socket.

18302 [EOPNOTSUPP] The operation is not supported for this socket's protocol.

18303 The *getsockname()* function may fail if:

18304 [EINVAL] The socket has been shut down.

18305 [ENOBUFS] Insufficient resources were available in the system to complete the function.

18306 **EXAMPLES**

18307 None.

18308 **APPLICATION USAGE**

18309 None.

18310 **RATIONALE**

18311 None.

18312 **FUTURE DIRECTIONS**

18313 None.

18314 **SEE ALSO**

18315 *accept()*, *bind()*, *getpeername()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-2001,
18316 <sys/socket.h>

18317 **CHANGE HISTORY**

18318 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

18319 The **restrict** keyword is added to the *getsockname()* prototype for alignment with the
18320 ISO/IEC 9899:1999 standard.

18321 **NAME**

18322 getsockopt — get the socket options

18323 **SYNOPSIS**

18324 #include <sys/socket.h>

```
18325       int getsockopt(int socket, int level, int option_name,
18326                      void *restrict option_value, socklen_t *restrict option_len);
```

18327 **DESCRIPTION**18328 The *getsockopt()* function manipulates options associated with a socket.

18329 The *getsockopt()* function shall retrieve the value for the option specified by the *option_name*
 18330 argument for the socket specified by the *socket* argument. If the size of the option value is greater
 18331 than *option_len*, the value stored in the object pointed to by the *option_value* argument shall be
 18332 silently truncated. Otherwise, the object pointed to by the *option_len* argument shall be modified
 18333 to indicate the actual length of the value.

18334 The *level* argument specifies the protocol level at which the option resides. To retrieve options at
 18335 the socket level, specify the *level* argument as SOL_SOCKET. To retrieve options at other levels,
 18336 supply the appropriate level identifier for the protocol controlling the option. For example, to
 18337 indicate that an option is interpreted by the TCP (Transmission Control Protocol), set *level* to
 18338 IPPROTO_TCP as defined in the <netinet/in.h> header.

18339 The socket in use may require the process to have appropriate privileges to use the *getsockopt()*
 18340 function.

18341 The *option_name* argument specifies a single option to be retrieved. It can be one of the following
 18342 values defined in <sys/socket.h>:

18343 SO_DEBUG Reports whether debugging information is being recorded. This option
 18344 shall store an **int** value. This is a Boolean option.

18345 SO_ACCEPTCONN Reports whether socket listening is enabled. This option shall store an **int**
 18346 value. This is a Boolean option.

18347 SO_BROADCAST Reports whether transmission of broadcast messages is supported, if this
 18348 is supported by the protocol. This option shall store an **int** value. This is a
 18349 Boolean option.

18350 SO_REUSEADDR Reports whether the rules used in validating addresses supplied to *bind()*
 18351 should allow reuse of local addresses, if this is supported by the protocol.
 18352 This option shall store an **int** value. This is a Boolean option.

18353 SO_KEEPALIVE Reports whether connections are kept active with periodic transmission
 18354 of messages, if this is supported by the protocol.

18355 If the connected socket fails to respond to these messages, the connection
 18356 shall be broken and threads writing to that socket shall be notified with a
 18357 SIGPIPE signal. This option shall store an **int** value. This is a Boolean
 18358 option.

18359 SO_LINGER Reports whether the socket lingers on *close()* if data is present. If
 18360 SO_LINGER is set, the system blocks the process during *close()* until it
 18361 can transmit the data or until the end of the interval indicated by the
 18362 *l_linger* member, whichever comes first. If SO_LINGER is not specified,
 18363 and *close()* is issued, the system handles the call in a way that allows the
 18364 process to continue as quickly as possible. This option shall store a **linger**
 18365 structure.

18366	SO_OOBINLINE	Reports whether the socket leaves received out-of-band data (data marked urgent) inline. This option shall store an int value. This is a Boolean option.
18367		
18368		
18369	SO_SNDBUF	Reports send buffer size information. This option shall store an int value.
18370	SO_RCVBUF	Reports receive buffer size information. This option shall store an int value.
18371		
18372	SO_ERROR	Reports information about error status and clears it. This option shall store an int value.
18373		
18374	SO_TYPE	Reports the socket type. This option shall store an int value. Socket types are described in Section 2.10.6 (on page 59).
18375		
18376	SO_DONTROUTE	Reports whether outgoing messages bypass the standard routing facilities. The destination shall be on a directly-connected network, and messages are directed to the appropriate network interface according to the destination address. The effect, if any, of this option depends on what protocol is in use. This option shall store an int value. This is a Boolean option.
18377		
18378		
18379		
18380		
18381		
18382	SO_RCVLOWAT	Reports the minimum number of bytes to process for socket input operations. The default value for SO_RCVLOWAT is 1. If SO_RCVLOWAT is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. (They may return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different from that returned; for example, out-of-band data.) This option shall store an int value. Note that not all implementations allow this option to be retrieved.
18383		
18384		
18385		
18386		
18387		
18388		
18389		
18390		
18391	SO_RCVTIMEO	Reports the timeout value for input operations. This option shall store a timeval structure with the number of seconds and microseconds specifying the limit on how long to wait for an input operation to complete. If a receive operation has blocked for this much time without receiving additional data, it shall return with a partial count or <i>errno</i> set to [EAGAIN] or [EWOULDBLOCK] if no data was received. The default for this option is zero, which indicates that a receive operation shall not time out. Note that not all implementations allow this option to be retrieved.
18392		
18393		
18394		
18395		
18396		
18397		
18398		
18399	SO_SNDLOWAT	Reports the minimum number of bytes to process for socket output operations. Non-blocking output operations shall process no data if flow control does not allow the smaller of the send low water mark value or the entire request to be processed. This option shall store an int value. Note that not all implementations allow this option to be retrieved.
18400		
18401		
18402		
18403		
18404	SO_SNDTIMEO	Reports the timeout value specifying the amount of time that an output function blocks because flow control prevents data from being sent. If a send operation has blocked for this time, it shall return with a partial count or with <i>errno</i> set to [EAGAIN] or [EWOULDBLOCK] if no data was sent. The default for this option is zero, which indicates that a send operation shall not time out. The option shall store a timeval structure. Note that not all implementations allow this option to be retrieved.
18405		
18406		
18407		
18408		
18409		
18410		
18411	For Boolean options, a zero value indicates that the option is disabled and a non-zero value	
18412	indicates that the option is enabled.	

18413 Options at other protocol levels vary in format and name.

18414 The socket in use may require the process to have appropriate privileges to use the *getsockopt()*
18415 function.

18416 **RETURN VALUE**

18417 Upon successful completion, *getsockopt()* shall return 0; otherwise, -1 shall be returned and *errno*
18418 set to indicate the error.

18419 **ERRORS**

18420 The *getsockopt()* function shall fail if:

18421 [EBADF] The *socket* argument is not a valid file descriptor.

18422 [EINVAL] The specified option is invalid at the specified socket level.

18423 [ENOPROTOOPT]

18424 The option is not supported by the protocol.

18425 [ENOTSOCK] The *socket* argument does not refer to a socket.

18426 The *getsockopt()* function may fail if:

18427 [EACCES] The calling process does not have the appropriate privileges.

18428 [EINVAL] The socket has been shut down.

18429 [ENOBUFS] Insufficient resources are available in the system to complete the function.

18430 **EXAMPLES**

18431 None.

18432 **APPLICATION USAGE**

18433 None.

18434 **RATIONALE**

18435 None.

18436 **FUTURE DIRECTIONS**

18437 None.

18438 **SEE ALSO**

18439 *bind()*, *close()*, *endprotoent()*, *setsockopt()*, *socket()*, the Base Definitions volume of
18440 IEEE Std 1003.1-2001, <sys/socket.h>, <netinet/in.h>

18441 **CHANGE HISTORY**

18442 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

18443 The **restrict** keyword is added to the *getsockopt()* prototype for alignment with the
18444 ISO/IEC 9899:1999 standard.

18445 **NAME**

18446 getsubopt — parse suboption arguments from a string

18447 **SYNOPSIS**

18448 XSI #include <stdlib.h>

18449 int getsubopt(char ***optionp*, char * const **tokens*, char ***valuep*);

18450

18451 **DESCRIPTION**18452 The *getsubopt()* function shall parse suboption arguments in a flag argument. Such options often
18453 result from the use of *getopt()*.18454 The *getsubopt()* argument *optionp* is a pointer to a pointer to the option argument string. The
18455 suboption arguments shall be separated by commas and each may consist of either a single
18456 token, or a token-value pair separated by an equal sign.18457 The *keylistp* argument shall be a pointer to a vector of strings. The end of the vector is identified
18458 by a null pointer. Each entry in the vector is one of the possible tokens that might be found in
18459 **optionp*. Since commas delimit suboption arguments in *optionp*, they should not appear in any of
18460 the strings pointed to by *keylistp*. Similarly, because an equal sign separates a token from its
18461 value, the application should not include an equal sign in any of the strings pointed to by
18462 *keylistp*.18463 The *valuep* argument is the address of a value string pointer.18464 If a comma appears in *optionp*, it shall be interpreted as a suboption separator. After commas
18465 have been processed, if there are one or more equal signs in a suboption string, the first equal
18466 sign in any suboption string shall be interpreted as a separator between a token and a value.
18467 Subsequent equal signs in a suboption string shall be interpreted as part of the value.18468 If the string at **optionp* contains only one suboption argument (equivalently, no commas),
18469 *getsubopt()* shall update **optionp* to point to the null character at the end of the string. Otherwise,
18470 it shall isolate the suboption argument by replacing the comma separator with a null character,
18471 and shall update **optionp* to point to the start of the next suboption argument. If the suboption
18472 argument has an associated value (equivalently, contains an equal sign), *getsubopt()* shall update
18473 **valuep* to point to the value's first character. Otherwise, it shall set **valuep* to a null pointer. The
18474 calling application may use this information to determine whether the presence or absence of a
18475 value for the suboption is an error.18476 Additionally, when *getsubopt()* fails to match the suboption argument with a token in the *keylistp*
18477 array, the calling application should decide if this is an error, or if the unrecognized option
18478 should be processed in another way.18479 **RETURN VALUE**18480 The *getsubopt()* function shall return the index of the matched token string, or -1 if no token
18481 strings were matched.18482 **ERRORS**

18483 No errors are defined.

18484 EXAMPLES

```

18485     #include <stdio.h>
18486     #include <stdlib.h>

18487     int do_all;
18488     const char *type;
18489     int read_size;
18490     int write_size;
18491     int read_only;

18492     enum
18493     {
18494         RO_OPTION = 0,
18495         RW_OPTION,
18496         READ_SIZE_OPTION,
18497         WRITE_SIZE_OPTION
18498     };

18499     const char *mount_opts[] =
18500     {
18501         [RO_OPTION] = "ro",
18502         [RW_OPTION] = "rw",
18503         [READ_SIZE_OPTION] = "rsize",
18504         [WRITE_SIZE_OPTION] = "wsize",
18505         NULL
18506     };

18507     int
18508     main(int argc, char *argv[])
18509     {
18510         char *subopts, *value;
18511         int opt;

18512         while ((opt = getopt(argc, argv, "at:o:")) != -1)
18513             switch(opt)
18514             {
18515                 case 'a':
18516                     do_all = 1;
18517                     break;
18518                 case 't':
18519                     type = optarg;
18520                     break;
18521                 case 'o':
18522                     subopts = optarg;
18523                     while (*subopts != '\0')
18524                         switch(getsubopt(&subopts, mount_opts, &value))
18525                         {
18526                             case RO_OPTION:
18527                                 read_only = 1;
18528                                 break;
18529                             case RW_OPTION:
18530                                 read_only = 0;
18531                                 break;
18532                             case READ_SIZE_OPTION:

```



```

18533         if (value == NULL)
18534             abort();
18535         read_size = atoi(value);
18536         break;
18537     case WRITE_SIZE_OPTION:
18538         if (value == NULL)
18539             abort();
18540         write_size = atoi(value);
18541         break;
18542     default:
18543         /* Unknown suboption. */
18544         printf("Unknown suboption '%s'\n", value);
18545         break;
18546     }
18547     break;
18548 default:
18549     abort();
18550 }
18551 /* Do the real work. */
18552 return 0;
18553 }

```

18554 Parsing Suboptions

18555 The following example uses the *getsubopt()* function to parse a *value* argument in the *optarg*
 18556 external variable returned by a call to *getopt()*.

```

18557 #include <stdlib.h>
18558 ...
18559 char *tokens[] = {"HOME", "PATH", "LOGNAME", (char *) NULL };
18560 char *value;
18561 int opt, index;
18562 while ((opt = getopt(argc, argv, "e:")) != -1) {
18563     switch(opt) {
18564         case 'e' :
18565             while ((index = getsubopt(&optarg, tokens, &value)) != -1) {
18566                 switch(index) {
18567                     ...
18568                 }
18569                 break;
18570             ...
18571         }
18572     }
18573     ...

```

18574 APPLICATION USAGE

18575 None.

18576 RATIONALE

18577 None.

18578 **FUTURE DIRECTIONS**

18579 None.

18580 **SEE ALSO**18581 *getopt()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>18582 **CHANGE HISTORY**

18583 First released in Issue 4, Version 2.

18584 **Issue 5**

18585 Moved from X/OPEN UNIX extension to BASE.

18586 **NAME**

18587 gettimeofday — get the date and time

18588 **SYNOPSIS**

18589 XSI #include <sys/time.h>

18590 int gettimeofday(struct timeval *restrict tp, void *restrict tzp);

18591

18592 **DESCRIPTION**

18593 The *gettimeofday()* function shall obtain the current time, expressed as seconds and
18594 microseconds since the Epoch, and store it in the **timeval** structure pointed to by *tp*. The
18595 resolution of the system clock is unspecified.

18596 If *tzp* is not a null pointer, the behavior is unspecified.

18597 **RETURN VALUE**

18598 The *gettimeofday()* function shall return 0 and no value shall be reserved to indicate an error.

18599 **ERRORS**

18600 No errors are defined.

18601 **EXAMPLES**

18602 None.

18603 **APPLICATION USAGE**

18604 None.

18605 **RATIONALE**

18606 None.

18607 **FUTURE DIRECTIONS**

18608 None.

18609 **SEE ALSO**

18610 *ctime()*, *ftime()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/time.h>

18611 **CHANGE HISTORY**

18612 First released in Issue 4, Version 2.

18613 **Issue 5**

18614 Moved from X/OPEN UNIX extension to BASE.

18615 **Issue 6**

18616 The DESCRIPTION is updated to refer to “seconds since the Epoch” rather than “seconds since
18617 00:00:00 UTC (Coordinated Universal Time), January 1 1970” for consistency with other *time*
18618 functions.

18619 The **restrict** keyword is added to the *gettimeofday()* prototype for alignment with the
18620 ISO/IEC 9899:1999 standard.

18621 **NAME**

18622 getuid — get a real user ID

18623 **SYNOPSIS**

18624 #include <unistd.h>

18625 uid_t getuid(void);

18626 **DESCRIPTION**18627 The *getuid()* function shall return the real user ID of the calling process.18628 **RETURN VALUE**18629 The *getuid()* function shall always be successful and no return value is reserved to indicate the
18630 error.18631 **ERRORS**

18632 No errors are defined.

18633 **EXAMPLES**18634 **Setting the Effective User ID to the Real User ID**18635 The following example sets the effective user ID and the real user ID of the current process to the
18636 real user ID of the caller.18637 #include <unistd.h>
18638 #include <sys/types.h>
18639 ...
18640 setreuid(getuid(), getuid());
18641 ...18642 **APPLICATION USAGE**

18643 None.

18644 **RATIONALE**

18645 None.

18646 **FUTURE DIRECTIONS**

18647 None.

18648 **SEE ALSO**18649 *getegid()*, *geteuid()*, *getgid()*, *setegid()*, *seteuid()*, *setgid()*, *setregid()*, *setreuid()*, *setuid()*, the Base
18650 Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>18651 **CHANGE HISTORY**

18652 First released in Issue 1. Derived from Issue 1 of the SVID.

18653 **Issue 6**

18654 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

18655 The following new requirements on POSIX implementations derive from alignment with the
18656 Single UNIX Specification:

- 18657
- The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
18658 required for conforming implementations of previous POSIX specifications, it was not
18659 required for UNIX applications.

18660 **NAME**

18661 getutxent, getutxid, getutxline — get user accounting database entries

18662 **SYNOPSIS**

18663 XSI #include <utmpx.h>

18664 struct utmpx *getutxent(void);

18665 struct utmpx *getutxid(const struct utmpx *id);

18666 struct utmpx *getutxline(const struct utmpx *line);

18667

18668 **DESCRIPTION**18669 Refer to *endutxent()*.

18670 **NAME**

18671 getwc — get a wide character from a stream

18672 **SYNOPSIS**

18673 #include <stdio.h>

18674 #include <wchar.h>

18675 wint_t getwc(FILE **stream*);18676 **DESCRIPTION**

18677 cx The functionality described on this reference page is aligned with the ISO C standard. Any
18678 conflict between the requirements described here and the ISO C standard is unintentional. This
18679 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

18680 The *getwc()* function shall be equivalent to *fgetwc()*, except that if it is implemented as a macro it
18681 may evaluate *stream* more than once, so the argument should never be an expression with side
18682 effects.

18683 **RETURN VALUE**18684 Refer to *fgetwc()*.18685 **ERRORS**18686 Refer to *fgetwc()*.18687 **EXAMPLES**

18688 None.

18689 **APPLICATION USAGE**

18690 Since it may be implemented as a macro, *getwc()* may treat incorrectly a *stream* argument with
18691 side effects. In particular, *getwc(*f++)* does not necessarily work as expected. Therefore, use of
18692 this function is not recommended; *fgetwc()* should be used instead.

18693 **RATIONALE**

18694 None.

18695 **FUTURE DIRECTIONS**

18696 None.

18697 **SEE ALSO**18698 *fgetwc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>, <wchar.h>18699 **CHANGE HISTORY**

18700 First released as a World-wide Portability Interface in Issue 4. Derived from the MSE working
18701 draft.

18702 **Issue 5**

18703 The Optional Header (OH) marking is removed from <stdio.h>.

18704 **NAME**

18705 getwchar — get a wide character from a stdin stream

18706 **SYNOPSIS**

18707 #include <wchar.h>

18708 wint_t getwchar(void);

18709 **DESCRIPTION**

18710 cx The functionality described on this reference page is aligned with the ISO C standard. Any
18711 conflict between the requirements described here and the ISO C standard is unintentional. This
18712 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

18713 The *getwchar()* function shall be equivalent to *getwc(stdin)*.18714 **RETURN VALUE**18715 Refer to *fgetwc()*.18716 **ERRORS**18717 Refer to *fgetwc()*.18718 **EXAMPLES**

18719 None.

18720 **APPLICATION USAGE**

18721 If the **wint_t** value returned by *getwchar()* is stored into a variable of type **wchar_t** and then
18722 compared against the **wint_t** macro WEOF, the result may be incorrect. Only the **wint_t** type is
18723 guaranteed to be able to represent any wide character and WEOF.

18724 **RATIONALE**

18725 None.

18726 **FUTURE DIRECTIONS**

18727 None.

18728 **SEE ALSO**18729 *fgetwc()*, *getwc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**wchar.h**>18730 **CHANGE HISTORY**

18731 First released as a World-wide Portability Interface in Issue 4. Derived from the MSE working
18732 draft.

18733 **NAME**

18734 getwd — get the current working directory pathname (**LEGACY**)

18735 **SYNOPSIS**

18736 XSI #include <unistd.h>

18737 char *getwd(char *path_name);

18738

18739 **DESCRIPTION**

18740 The *getwd()* function shall determine an absolute pathname of the current working directory of
18741 the calling process, and copy a string containing that pathname into the array pointed to by the
18742 *path_name* argument.

18743 If the length of the pathname of the current working directory is greater than ({PATH_MAX}+1)
18744 including the null byte, *getwd()* shall fail and return a null pointer.

18745 **RETURN VALUE**

18746 Upon successful completion, a pointer to the string containing the absolute pathname of the
18747 current working directory shall be returned. Otherwise, *getwd()* shall return a null pointer and
18748 the contents of the array pointed to by *path_name* are undefined.

18749 **ERRORS**

18750 No errors are defined.

18751 **EXAMPLES**

18752 None.

18753 **APPLICATION USAGE**

18754 For applications portability, the *getcwd()* function should be used to determine the current
18755 working directory instead of *getwd()*.

18756 **RATIONALE**

18757 Since the user cannot specify the length of the buffer passed to *getwd()*, use of this function is
18758 discouraged. The length of a pathname described in {PATH_MAX} is file system-dependent and
18759 may vary from one mount point to another, or might even be unlimited. It is possible to
18760 overflow this buffer in such a way as to cause applications to fail, or possible system security
18761 violations.

18762 It is recommended that the *getcwd()* function should be used to determine the current working
18763 directory.

18764 **FUTURE DIRECTIONS**

18765 This function may be withdrawn in a future version.

18766 **SEE ALSO**

18767 *getcwd()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

18768 **CHANGE HISTORY**

18769 First released in Issue 4, Version 2.

18770 **Issue 5**

18771 Moved from X/OPEN UNIX extension to BASE.

18772 **Issue 6**

18773 This function is marked LEGACY.

18774 **NAME**

18775 glob, globfree — generate pathnames matching a pattern

18776 **SYNOPSIS**

18777 #include <glob.h>

```
18778 int glob(const char *restrict pattern, int flags,
18779         int (*errfunc)(const char *epath, int eerrno),
18780         glob_t *restrict pglob);
18781 void globfree(glob_t *pglob);
```

18782 **DESCRIPTION**

18783 The *glob()* function is a pathname generator that shall implement the rules defined in the Shell
 18784 and Utilities volume of IEEE Std 1003.1-2001, Section 2.13, Pattern Matching Notation, with
 18785 optional support for rule 3 in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section
 18786 2.13.3, Patterns Used for Filename Expansion.

18787 The structure type **glob_t** is defined in <glob.h> and includes at least the following members:

18788

18789

18790

18791

18792

Member Type	Member Name	Description
size_t	gl_pathc	Count of paths matched by <i>pattern</i> .
char **	gl_pathv	Pointer to a list of matched pathnames.
size_t	gl_offs	Slots to reserve at the beginning of <i>gl_pathv</i> .

18793 The argument *pattern* is a pointer to a pathname pattern to be expanded. The *glob()* function
 18794 shall match all accessible pathnames against this pattern and develop a list of all pathnames that
 18795 match. In order to have access to a pathname, *glob()* requires search permission on every
 18796 component of a path except the last, and read permission on each directory of any filename
 18797 component of *pattern* that contains any of the following special characters: '*', '?', and '['.

18798 The *glob()* function shall store the number of matched pathnames into *pglob->gl_pathc* and a
 18799 pointer to a list of pointers to pathnames into *pglob->gl_pathv*. The pathnames shall be in sort
 18800 order as defined by the current setting of the *LC_COLLATE* category; see the Base Definitions
 18801 volume of IEEE Std 1003.1-2001, Section 7.3.2, *LC_COLLATE*. The first pointer after the last
 18802 pathname shall be a null pointer. If the pattern does not match any pathnames, the returned
 18803 number of matched paths is set to 0, and the contents of *pglob->gl_pathv* are implementation-
 18804 defined.

18805 It is the caller's responsibility to create the structure pointed to by *pglob*. The *glob()* function shall
 18806 allocate other space as needed, including the memory pointed to by *gl_pathv*. The *globfree()*
 18807 function shall free any space associated with *pglob* from a previous call to *glob()*.

18808 The *flags* argument is used to control the behavior of *glob()*. The value of *flags* is a bitwise-
 18809 inclusive OR of zero or more of the following constants, which are defined in <glob.h>:

18810	GLOBAL_APPEND	Append pathnames generated to the ones from a previous call to <i>glob()</i> .
18811	GLOBAL_DOOFFS	Make use of <i>pglob->gl_offs</i> . If this flag is set, <i>pglob->gl_offs</i> is used to
18812		specify how many null pointers to add to the beginning of
18813		<i>pglob->gl_pathv</i> . In other words, <i>pglob->gl_pathv</i> shall point to
18814		<i>pglob->gl_offs</i> null pointers, followed by <i>pglob->gl_pathc</i> pathname
18815		pointers, followed by a null pointer.
18816	GLOBAL_ERR	Cause <i>glob()</i> to return when it encounters a directory that it cannot open
18817		or read. Ordinarily, <i>glob()</i> continues to find matches.

18818	GLOB_MARK	Each pathname that is a directory that matches <i>pattern</i> shall have a slash appended.
18819		
18820	GLOB_NOCHECK	Supports rule 3 in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.13.3, Patterns Used for Filename Expansion. If <i>pattern</i> does not match any pathname, then <i>glob()</i> shall return a list consisting of only <i>pattern</i> , and the number of matched pathnames is 1.
18821		
18822		
18823		
18824	GLOB_NOESCAPE	Disable backslash escaping.
18825	GLOB_NOSORT	Ordinarily, <i>glob()</i> sorts the matching pathnames according to the current setting of the <i>LC_COLLATE</i> category; see the Base Definitions volume of IEEE Std 1003.1-2001, Section 7.3.2, <i>LC_COLLATE</i> . When this flag is used, the order of pathnames returned is unspecified.
18826		
18827		
18828		
18829	The GLOB_APPEND flag can be used to append a new set of pathnames to those found in a previous call to <i>glob()</i> . The following rules apply to applications when two or more calls to <i>glob()</i> are made with the same value of <i>pglob</i> and without intervening calls to <i>globfree()</i> :	
18830		
18831		
18832	1.	The first such call shall not set GLOB_APPEND. All subsequent calls shall set it.
18833	2.	All the calls shall set GLOB_DOOFFS, or all shall not set it.
18834	3.	After the second call, <i>pglob->gl_pathv</i> points to a list containing the following:
18835	a.	Zero or more null pointers, as specified by GLOB_DOOFFS and <i>pglob->gl_offs</i> .
18836	b.	Pointers to the pathnames that were in the <i>pglob->gl_pathv</i> list before the call, in the same order as before.
18837		
18838	c.	Pointers to the new pathnames generated by the second call, in the specified order.
18839	4.	The count returned in <i>pglob->gl_pathc</i> shall be the total number of pathnames from the two calls.
18840		
18841	5.	The application can change any of the fields after a call to <i>glob()</i> . If it does, the application shall reset them to the original value before a subsequent call, using the same <i>pglob</i> value, to <i>globfree()</i> or <i>glob()</i> with the GLOB_APPEND flag.
18842		
18843		
18844	If, during the search, a directory is encountered that cannot be opened or read and <i>errfunc</i> is not a null pointer, <i>glob()</i> calls (<i>*errfunc()</i>) with two arguments:	
18845		
18846	1.	The <i>epath</i> argument is a pointer to the path that failed.
18847	2.	The <i>errno</i> argument is the value of <i>errno</i> from the failure, as set by <i>opendir()</i> , <i>readdir()</i> , or <i>stat()</i> . (Other values may be used to report other errors not explicitly documented for those functions.)
18848		
18849		
18850	If (<i>*errfunc()</i>) is called and returns non-zero, or if the GLOB_ERR flag is set in <i>flags</i> , <i>glob()</i> shall stop the scan and return GLOB_ABORTED after setting <i>gl_pathc</i> and <i>gl_pathv</i> in <i>pglob</i> to reflect the paths already scanned. If GLOB_ERR is not set and either <i>errfunc</i> is a null pointer or (<i>*errfunc()</i>) returns 0, the error shall be ignored.	
18851		
18852		
18853		
18854	The <i>glob()</i> function shall not fail because of large files.	
18855	RETURN VALUE	
18856	Upon successful completion, <i>glob()</i> shall return 0. The argument <i>pglob->gl_pathc</i> shall return the number of matched pathnames and the argument <i>pglob->gl_pathv</i> shall contain a pointer to a null-terminated list of matched and sorted pathnames. However, if <i>pglob->gl_pathc</i> is 0, the content of <i>pglob->gl_pathv</i> is undefined.	
18857		
18858		
18859		

18860 The *globfree()* function shall not return a value.

18861 If *glob()* terminates due to an error, it shall return one of the non-zero constants defined in
18862 *<glob.h>*. The arguments *pglob->gl_pathc* and *pglob->gl_pathv* are still set as defined above.

18863 ERRORS

18864 The *glob()* function shall fail and return the corresponding value if:

18865 GLOB_ABORTED The scan was stopped because GLOB_ERR was set or (*errfunc())
18866 returned non-zero.

18867 GLOB_NOMATCH The pattern does not match any existing pathname, and
18868 GLOB_NOCHECK was not set in flags.

18869 GLOB_NOSPACE An attempt to allocate memory failed.

18870 EXAMPLES

18871 One use of the GLOB_DOOFFS flag is by applications that build an argument list for use with
18872 *execv()*, *execve()*, or *execvp()*. Suppose, for example, that an application wants to do the
18873 equivalent of:

```
18874 ls -l *.c
```

18875 but for some reason:

```
18876 system("ls -l *.c")
```

18877 is not acceptable. The application could obtain approximately the same result using the
18878 sequence:

```
18879 globbuf.gl_offs = 2;  
18880 glob("*.c", GLOB_DOOFFS, NULL, &globbuf);  
18881 globbuf.gl_pathv[0] = "ls";  
18882 globbuf.gl_pathv[1] = "-l";  
18883 execvp("ls", &globbuf.gl_pathv[0]);
```

18884 Using the same example:

```
18885 ls -l *.c *.h
```

18886 could be approximately simulated using GLOB_APPEND as follows:

```
18887 globbuf.gl_offs = 2;  
18888 glob("*.c", GLOB_DOOFFS, NULL, &globbuf);  
18889 glob("*.h", GLOB_DOOFFS|GLOB_APPEND, NULL, &globbuf);  
18890 ...
```

18891 APPLICATION USAGE

18892 This function is not provided for the purpose of enabling utilities to perform pathname
18893 expansion on their arguments, as this operation is performed by the shell, and utilities are
18894 explicitly not expected to redo this. Instead, it is provided for applications that need to do
18895 pathname expansion on strings obtained from other sources, such as a pattern typed by a user or
18896 read from a file.

18897 If a utility needs to see if a pathname matches a given pattern, it can use *fnmatch()*.

18898 Note that *gl_pathc* and *gl_pathv* have meaning even if *glob()* fails. This allows *glob()* to report
18899 partial results in the event of an error. However, if *gl_pathc* is 0, *gl_pathv* is unspecified even if
18900 *glob()* did not return an error.

18901 The GLOB_NOCHECK option could be used when an application wants to expand a pathname
18902 if wildcards are specified, but wants to treat the pattern as just a string otherwise. The *sh* utility

18903 might use this for option-arguments, for example.

18904 The new pathnames generated by a subsequent call with GLOB_APPEND are not sorted
 18905 together with the previous pathnames. This mirrors the way that the shell handles pathname
 18906 expansion when multiple expansions are done on a command line.

18907 Applications that need tilde and parameter expansion should use *wordexp()*.

18908 RATIONALE

18909 It was claimed that the GLOB_DOOFFS flag is unnecessary because it could be simulated using:

```
18910 new = (char **)malloc((n + pglob->gl_pathc + 1)
18911     * sizeof(char *));
18912 (void) memcpy(new+n, pglob->gl_pathv,
18913     pglob->gl_pathc * sizeof(char *));
18914 (void) memset(new, 0, n * sizeof(char *));
18915 free(pglob->gl_pathv);
18916 pglob->gl_pathv = new;
```

18917 However, this assumes that the memory pointed to by *gl_pathv* is a block that was separately
 18918 created using *malloc()*. This is not necessarily the case. An application should make no
 18919 assumptions about how the memory referenced by fields in *pglob* was allocated. It might have
 18920 been obtained from *malloc()* in a large chunk and then carved up within *glob()*, or it might have
 18921 been created using a different memory allocator. It is not the intent of the standard developers to
 18922 specify or imply how the memory used by *glob()* is managed.

18923 The GLOB_APPEND flag would be used when an application wants to expand several different
 18924 patterns into a single list.

18925 FUTURE DIRECTIONS

18926 None.

18927 SEE ALSO

18928 *exec*, *fnmatch()*, *opendir()*, *readdir()*, *stat()*, *wordexp()*, the Base Definitions volume of
 18929 IEEE Std 1003.1-2001, <**glob.h**>, the Shell and Utilities volume of IEEE Std 1003.1-2001

18930 CHANGE HISTORY

18931 First released in Issue 4. Derived from the ISO POSIX-2 standard.

18932 Issue 5

18933 Moved from POSIX2 C-language Binding to BASE.

18934 Issue 6

18935 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

18936 The **restrict** keyword is added to the *glob()* prototype for alignment with the ISO/IEC 9899:1999
 18937 standard.

18938 **NAME**

18939 gmtime, gmtime_r — convert a time value to a broken-down UTC time

18940 **SYNOPSIS**

18941 #include <time.h>

18942 struct tm *gmtime(const time_t *timer);

18943 TSF struct tm *gmtime_r(const time_t *restrict timer,

18944 struct tm *restrict result);

18945

18946 **DESCRIPTION**

18947 CX For *gmtime()*: The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

18950 The *gmtime()* function shall convert the time in seconds since the Epoch pointed to by *timer* into a broken-down time, expressed as Coordinated Universal Time (UTC).

18952 CX The relationship between a time in seconds since the Epoch used as an argument to *gmtime()* and the **tm** structure (defined in the <time.h> header) is that the result shall be as specified in the expression given in the definition of seconds since the Epoch (see the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.14, Seconds Since the Epoch), where the names in the structure and in the expression correspond.

18957 TSF The same relationship shall apply for *gmtime_r()*.

18958 CX The *gmtime()* function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

18960 The *asctime()*, *ctime()*, *gmtime()*, and *localtime()* functions shall return values in one of two static objects: a broken-down time structure and an array of type **char**. Execution of any of the functions may overwrite the information returned in either of these objects by any of the other functions.

18964 TSF The *gmtime_r()* function shall convert the time in seconds since the Epoch pointed to by *timer* into a broken-down time expressed as Coordinated Universal Time (UTC). The broken-down time is stored in the structure referred to by *result*. The *gmtime_r()* function shall also return the address of the same structure.

18968 **RETURN VALUE**

18969 The *gmtime()* function shall return a pointer to a **struct tm**.

18970 TSF Upon successful completion, *gmtime_r()* shall return the address of the structure pointed to by the argument *result*. If an error is detected, or UTC is not available, *gmtime_r()* shall return a null pointer.

18973 **ERRORS**

18974 No errors are defined.

18975 **EXAMPLES**

18976 None.

18977 **APPLICATION USAGE**

18978 The *gmtime_r()* function is thread-safe and returns values in a user-supplied buffer instead of
18979 possibly using a static data area that may be overwritten by each call.

18980 **RATIONALE**

18981 None.

18982 **FUTURE DIRECTIONS**

18983 None.

18984 **SEE ALSO**

18985 *asctime()*, *clock()*, *ctime()*, *difftime()*, *localtime()*, *mktime()*, *strftime()*, *strptime()*, *time()*, *utime()*,
18986 the Base Definitions volume of IEEE Std 1003.1-2001, <**time.h**>

18987 **CHANGE HISTORY**

18988 First released in Issue 1. Derived from Issue 1 of the SVID.

18989 **Issue 5**

18990 A note indicating that the *gmtime()* function need not be reentrant is added to the
18991 DESCRIPTION.

18992 The *gmtime_r()* function is included for alignment with the POSIX Threads Extension.18993 **Issue 6**18994 The *gmtime_r()* function is marked as part of the Thread-Safe Functions option.

18995 Extensions beyond the ISO C standard are marked.

18996 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
18997 its avoidance of possibly using a static data area.

18998 The **restrict** keyword is added to the *gmtime_r()* prototype for alignment with the
18999 ISO/IEC 9899:1999 standard.

19000 NAME

19001 grantpt — grant access to the slave pseudo-terminal device

19002 SYNOPSIS

19003 XSI #include <stdlib.h>

19004 int grantpt(int *fildev*);

19005

19006 DESCRIPTION

19007 The *grantpt()* function shall change the mode and ownership of the slave pseudo-terminal device associated with its master pseudo-terminal counterpart. The *fildev* argument is a file descriptor that refers to a master pseudo-terminal device. The user ID of the slave shall be set to the real UID of the calling process and the group ID shall be set to an unspecified group ID. The permission mode of the slave pseudo-terminal shall be set to readable and writable by the owner, and writable by the group.

19013 The behavior of the *grantpt()* function is unspecified if the application has installed a signal handler to catch SIGCHLD signals.

19015 RETURN VALUE

19016 Upon successful completion, *grantpt()* shall return 0; otherwise, it shall return -1 and set *errno* to indicate the error.

19018 ERRORS

19019 The *grantpt()* function may fail if:

19020 [EBADF] The *fildev* argument is not a valid open file descriptor.

19021 [EINVAL] The *fildev* argument is not associated with a master pseudo-terminal device.

19022 [EACCES] The corresponding slave pseudo-terminal device could not be accessed.

19023 EXAMPLES

19024 None.

19025 APPLICATION USAGE

19026 None.

19027 RATIONALE

19028 None.

19029 FUTURE DIRECTIONS

19030 None.

19031 SEE ALSO

19032 *open()*, *ptsname()*, *unlockpt()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

19033 CHANGE HISTORY

19034 First released in Issue 4, Version 2.

19035 Issue 5

19036 Moved from X/OPEN UNIX extension to BASE.

19037 The last paragraph of the DESCRIPTION is moved from the APPLICATION USAGE section.

19038 **NAME**

19039 h_errno — error return value for network database operations

19040 **SYNOPSIS**

19041 OB #include <netdb.h>

19042

19043 **DESCRIPTION**

19044 This method of returning errors is used only in connection with obsolescent functions.

19045 The <netdb.h> header provides a declaration of *h_errno* as a modifiable lvalue of type **int**.

19046 It is unspecified whether *h_errno* is a macro or an identifier declared with external linkage. If a
19047 macro definition is suppressed in order to access an actual object, or a program defines an
19048 identifier with the name *h_errno*, the behavior is undefined.

19049 **RETURN VALUE**

19050 None.

19051 **ERRORS**

19052 No errors are defined.

19053 **EXAMPLES**

19054 None.

19055 **APPLICATION USAGE**

19056 Applications should obtain the definition of *h_errno* by the inclusion of the <netdb.h> header.

19057 **RATIONALE**

19058 None.

19059 **FUTURE DIRECTIONS**

19060 *h_errno* may be withdrawn in a future version.

19061 **SEE ALSO**

19062 *endhostent()*, *errno*, the Base Definitions volume of IEEE Std 1003.1-2001, <netdb.h>

19063 **CHANGE HISTORY**

19064 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19065 **NAME**

19066 hcreate, hdestroy, hsearch — manage hash search table

19067 **SYNOPSIS**

```

19068 xSI      #include <search.h>

19069          int hcreate(size_t nel);
19070          void hdestroy(void);
19071          ENTRY *hsearch(ENTRY item, ACTION action);
19072

```

19073 **DESCRIPTION**19074 The *hcreate()*, *hdestroy()*, and *hsearch()* functions shall manage hash search tables.

19075 The *hcreate()* function shall allocate sufficient space for the table, and the application shall ensure it is called before *hsearch()* is used. The *nel* argument is an estimate of the maximum number of entries that the table shall contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

19079 The *hdestroy()* function shall dispose of the search table, and may be followed by another call to *hcreate()*. After the call to *hdestroy()*, the data can no longer be considered accessible.

19081 The *hsearch()* function is a hash-table search routine. It shall return a pointer into a hash table indicating the location at which an entry can be found. The *item* argument is a structure of type **ENTRY** (defined in the *<search.h>* header) containing two pointers: *item.key* points to the comparison key (a **char ***), and *item.data* (a **void ***) points to any other data to be associated with that key. The comparison function used by *hsearch()* is *strcmp()*. The *action* argument is a member of an enumeration type **ACTION** indicating the disposition of the entry if it cannot be found in the table. **ENTER** indicates that the item should be inserted in the table at an appropriate point. **FIND** indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a null pointer.

19090 These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

19092 **RETURN VALUE**

19093 The *hcreate()* function shall return 0 if it cannot allocate sufficient space for the table; otherwise, it shall return non-zero.

19095 The *hdestroy()* function shall not return a value.

19096 The *hsearch()* function shall return a null pointer if either the action is **FIND** and the item could not be found or the action is **ENTER** and the table is full.

19098 **ERRORS**19099 The *hcreate()* and *hsearch()* functions may fail if:

19100 [ENOMEM] Insufficient storage space is available.

19101 **EXAMPLES**

19102 The following example reads in strings followed by two numbers and stores them in a hash table, discarding duplicates. It then reads in strings and finds the matching entry in the hash table and prints it out.

```

19105          #include <stdio.h>
19106          #include <search.h>
19107          #include <string.h>

19108          struct info {          /* This is the info stored in the table */
19109              int age, room;      /* other than the key. */

```



```

19110     };
19111     #define NUM_EMPL    5000    /* # of elements in search table. */
19112     int main(void)
19113     {
19114         char string_space[NUM_EMPL*20];    /* Space to store strings. */
19115         struct info info_space[NUM_EMPL]; /* Space to store employee info. */
19116         char *str_ptr = string_space;      /* Next space in string_space. */
19117         struct info *info_ptr = info_space;
19118                                         /* Next space in info_space. */
19119         ENTRY item;
19120         ENTRY *found_item; /* Name to look for in table. */
19121         char name_to_find[30];
19122         int i = 0;
19123         /* Create table; no error checking is performed. */
19124         (void) hcreate(NUM_EMPL);
19125         while (scanf("%s%d%d", str_ptr, &info_ptr->age,
19126                     &info_ptr->room) != EOF && i++ < NUM_EMPL) {
19127             /* Put information in structure, and structure in item. */
19128             item.key = str_ptr;
19129             item.data = info_ptr;
19130             str_ptr += strlen(str_ptr) + 1;
19131             info_ptr++;
19132             /* Put item into table. */
19133             (void) hsearch(item, ENTER);
19134         }
19135         /* Access table. */
19136         item.key = name_to_find;
19137         while (scanf("%s", item.key) != EOF) {
19138             if ((found_item = hsearch(item, FIND)) != NULL) {
19139                 /* If item is in the table. */
19140                 (void)printf("found %s, age = %d, room = %d\n",
19141                             found_item->key,
19142                             ((struct info *)found_item->data)->age,
19143                             ((struct info *)found_item->data)->room);
19144             } else
19145                 (void)printf("no such employee %s\n", name_to_find);
19146         }
19147         return 0;
19148     }

```

19149 APPLICATION USAGE

19150 The *hcreate()* and *hsearch()* functions may use *malloc()* to allocate space.

19151 RATIONALE

19152 None.

19153 **FUTURE DIRECTIONS**

19154 None.

19155 **SEE ALSO**

19156 *bsearch()*, *lsearch()*, *malloc()*, *strcmp()*, *tsearch()*, the Base Definitions volume of
19157 IEEE Std 1003.1-2001, <**search.h**>

19158 **CHANGE HISTORY**

19159 First released in Issue 1. Derived from Issue 1 of the SVID.

19160 **Issue 6**

19161 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

19162 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

19163 **NAME**

19164 htonl, htons, ntohl, ntohs — convert values between host and network byte order

19165 **SYNOPSIS**

19166 #include <arpa/inet.h>

19167 uint32_t htonl(uint32_t *hostlong*);

19168 uint16_t htons(uint16_t *hostshort*);

19169 uint32_t ntohl(uint32_t *netlong*);

19170 uint16_t ntohs(uint16_t *netshort*);

19171 **DESCRIPTION**

19172 These functions shall convert 16-bit and 32-bit quantities between network byte order and host
19173 byte order.

19174 On some implementations, these functions are defined as macros.

19175 The **uint32_t** and **uint16_t** types are defined in <inttypes.h>.

19176 **RETURN VALUE**

19177 The *htonl()* and *htons()* functions shall return the argument value converted from host to
19178 network byte order.

19179 The *ntohl()* and *ntohs()* functions shall return the argument value converted from network to
19180 host byte order.

19181 **ERRORS**

19182 No errors are defined.

19183 **EXAMPLES**

19184 None.

19185 **APPLICATION USAGE**

19186 These functions are most often used in conjunction with IPv4 addresses and ports as returned by
19187 *gethostent()* and *getservent()*.

19188 **RATIONALE**

19189 None.

19190 **FUTURE DIRECTIONS**

19191 None.

19192 **SEE ALSO**

19193 *endhostent()*, *endservent()*, the Base Definitions volume of IEEE Std 1003.1-2001, <inttypes.h>,
19194 <arpa/inet.h>

19195 **CHANGE HISTORY**

19196 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19197 **NAME**

19198 hypot, hypotf, hypotl — Euclidean distance function

19199 **SYNOPSIS**

19200 #include <math.h>

19201 double hypot(double x, double y);

19202 float hypotf(float x, float y);

19203 long double hypotl(long double x, long double y);

19204 **DESCRIPTION**

19205 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 19206 conflict between the requirements described here and the ISO C standard is unintentional. This
 19207 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

19208 These functions shall compute the value of the square root of x^2+y^2 without undue overflow or
 19209 underflow.

19210 An application wishing to check for error situations should set *errno* to zero and call
 19211 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 19212 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 19213 zero, an error has occurred.

19214 **RETURN VALUE**

19215 Upon successful completion, these functions shall return the length of the hypotenuse of a
 19216 right-angled triangle with sides of length *x* and *y*.

19217 If the correct value would cause overflow, a range error shall occur and *hypot()*, *hypotf()*, and
 19218 *hypotl()* shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL,
 19219 respectively.

19220 **MX** If *x* or *y* is $\pm\text{Inf}$, $+\text{Inf}$ shall be returned (even if one of *x* or *y* is NaN).

19221 If *x* or *y* is NaN, and the other is not $\pm\text{Inf}$, a NaN shall be returned.

19222 If both arguments are subnormal and the correct result is subnormal, a range error may occur
 19223 and the correct result is returned.

19224 **ERRORS**

19225 These functions shall fail if:

19226 Range Error The result overflows.

19227 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 19228 then *errno* shall be set to [ERANGE]. If the integer expression
 19229 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow
 19230 floating-point exception shall be raised.

19231 These functions may fail if:

19232 **MX** Range Error The result underflows.

19233 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 19234 then *errno* shall be set to [ERANGE]. If the integer expression
 19235 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
 19236 floating-point exception shall be raised.

19237 **EXAMPLES**

19238 None.

19239 **APPLICATION USAGE**19240 *hypot(x,y)*, *hypot(y,x)*, and *hypot(x, -y)* are equivalent.19241 *hypot(x, ±0)* is equivalent to *fabs(x)*.19242 Underflow only happens when both *x* and *y* are subnormal and the (inexact) result is also
19243 subnormal.19244 These functions take precautions against overflow during intermediate steps of the
19245 computation.19246 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
19247 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.19248 **RATIONALE**

19249 None.

19250 **FUTURE DIRECTIONS**

19251 None.

19252 **SEE ALSO**19253 *feclearexcept()*, *fetestexcept()*, *isnan()*, *sqrt()*, the Base Definitions volume of IEEE Std 1003.1-2001,
19254 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>19255 **CHANGE HISTORY**

19256 First released in Issue 1. Derived from Issue 1 of the SVID.

19257 **Issue 5**19258 The DESCRIPTION is updated to indicate how an application should check for an error. This
19259 text was previously published in the APPLICATION USAGE section.19260 **Issue 6**19261 The *hypot()* function is no longer marked as an extension.19262 The *hypotf()* and *hypotl()* functions are added for alignment with the ISO/IEC 9899:1999
19263 standard.19264 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
19265 revised to align with the ISO/IEC 9899:1999 standard.19266 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
19267 marked.

19268 **NAME**

19269 iconv — codeset conversion function

19270 **SYNOPSIS**

```

19271 xSI      #include <iconv.h>

19272      size_t iconv(iconv_t cd, char **restrict inbuf,
19273                  size_t *restrict inbytesleft, char **restrict outbuf,
19274                  size_t *restrict outbytesleft);
19275

```

19276 **DESCRIPTION**

19277 The *iconv()* function shall convert the sequence of characters from one codeset, in the array
 19278 specified by *inbuf*, into a sequence of corresponding characters in another codeset, in the array
 19279 specified by *outbuf*. The codesets are those specified in the *iconv_open()* call that returned the
 19280 conversion descriptor, *cd*. The *inbuf* argument points to a variable that points to the first
 19281 character in the input buffer and *inbytesleft* indicates the number of bytes to the end of the buffer
 19282 to be converted. The *outbuf* argument points to a variable that points to the first available byte in
 19283 the output buffer and *outbytesleft* indicates the number of the available bytes to the end of the
 19284 buffer.

19285 For state-dependent encodings, the conversion descriptor *cd* is placed into its initial shift state by
 19286 a call for which *inbuf* is a null pointer, or for which *inbuf* points to a null pointer. When *iconv()*
 19287 is called in this way, and if *outbuf* is not a null pointer or a pointer to a null pointer, and *outbytesleft*
 19288 points to a positive value, *iconv()* shall place, into the output buffer, the byte sequence to change
 19289 the output buffer to its initial shift state. If the output buffer is not large enough to hold the
 19290 entire reset sequence, *iconv()* shall fail and set *errno* to [E2BIG]. Subsequent calls with *inbuf* as
 19291 other than a null pointer or a pointer to a null pointer cause the conversion to take place from
 19292 the current state of the conversion descriptor.

19293 If a sequence of input bytes does not form a valid character in the specified codeset, conversion
 19294 shall stop after the previous successfully converted character. If the input buffer ends with an
 19295 incomplete character or shift sequence, conversion shall stop after the previous successfully
 19296 converted bytes. If the output buffer is not large enough to hold the entire converted input,
 19297 conversion shall stop just prior to the input bytes that would cause the output buffer to
 19298 overflow. The variable pointed to by *inbuf* shall be updated to point to the byte following the last
 19299 byte successfully used in the conversion. The value pointed to by *inbytesleft* shall be
 19300 decremented to reflect the number of bytes still not converted in the input buffer. The variable
 19301 pointed to by *outbuf* shall be updated to point to the byte following the last byte of converted
 19302 output data. The value pointed to by *outbytesleft* shall be decremented to reflect the number of
 19303 bytes still available in the output buffer. For state-dependent encodings, the conversion
 19304 descriptor shall be updated to reflect the shift state in effect at the end of the last successfully
 19305 converted byte sequence.

19306 If *iconv()* encounters a character in the input buffer that is valid, but for which an identical
 19307 character does not exist in the target codeset, *iconv()* shall perform an implementation-defined
 19308 conversion on this character.

19309 **RETURN VALUE**

19310 The *iconv()* function shall update the variables pointed to by the arguments to reflect the extent
 19311 of the conversion and return the number of non-identical conversions performed. If the entire
 19312 string in the input buffer is converted, the value pointed to by *inbytesleft* shall be 0. If the input
 19313 conversion is stopped due to any conditions mentioned above, the value pointed to by *inbytesleft*
 19314 shall be non-zero and *errno* shall be set to indicate the condition. If an error occurs, *iconv()* shall
 19315 return (**size_t**)-1 and set *errno* to indicate the error.

19316 **ERRORS**19317 The *iconv()* function shall fail if:

19318 [EILSEQ] Input conversion stopped due to an input byte that does not belong to the
 19319 input codeset.

19320 [E2BIG] Input conversion stopped due to lack of space in the output buffer.

19321 [EINVAL] Input conversion stopped due to an incomplete character or shift sequence at
 19322 the end of the input buffer.

19323 The *iconv()* function may fail if:

19324 [EBADF] The *cd* argument is not a valid open conversion descriptor.

19325 **EXAMPLES**

19326 None.

19327 **APPLICATION USAGE**

19328 The *inbuf* argument indirectly points to the memory area which contains the conversion input
 19329 data. The *outbuf* argument indirectly points to the memory area which is to contain the result of
 19330 the conversion. The objects indirectly pointed to by *inbuf* and *outbuf* are not restricted to
 19331 containing data that is directly representable in the ISO C standard language **char** data type. The
 19332 type of *inbuf* and *outbuf*, **char ****, does not imply that the objects pointed to are interpreted as
 19333 null-terminated C strings or arrays of characters. Any interpretation of a byte sequence that
 19334 represents a character in a given character set encoding scheme is done internally within the
 19335 codeset converters. For example, the area pointed to indirectly by *inbuf* and/or *outbuf* can
 19336 contain all zero octets that are not interpreted as string terminators but as coded character data
 19337 according to the respective codeset encoding scheme. The type of the data (**char**, **short**, **long**, and
 19338 so on) read or stored in the objects is not specified, but may be inferred for both the input and
 19339 output data by the converters determined by the *fromcode* and *toencode* arguments of *iconv_open()*.

19340 Regardless of the data type inferred by the converter, the size of the remaining space in both
 19341 input and output objects (the *inbytesleft* and *outbytesleft* arguments) is always measured in bytes.

19342 For implementations that support the conversion of state-dependent encodings, the conversion
 19343 descriptor must be able to accurately reflect the shift-state in effect at the end of the last
 19344 successful conversion. It is not required that the conversion descriptor itself be updated, which
 19345 would require it to be a pointer type. Thus, implementations are free to implement the
 19346 descriptor as a handle (other than a pointer type) by which the conversion information can be
 19347 accessed and updated.

19348 **RATIONALE**

19349 None.

19350 **FUTURE DIRECTIONS**

19351 None.

19352 **SEE ALSO**19353 *iconv_open()*, *iconv_close()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**iconv.h**>19354 **CHANGE HISTORY**

19355 First released in Issue 4. Derived from the HP-UX Manual.

19356 **Issue 6**19357 The SYNOPSIS has been corrected to align with the <**iconv.h**> reference page.

19358 The **restrict** keyword is added to the *iconv()* prototype for alignment with the
 19359 ISO/IEC 9899:1999 standard.

19360 **NAME**

19361 iconv_close — codeset conversion deallocation function

19362 **SYNOPSIS**19363 XSI `#include <iconv.h>`19364 `int iconv_close(iconv_t cd);`

19365

19366 **DESCRIPTION**19367 The *iconv_close()* function shall deallocate the conversion descriptor *cd* and all other associated
19368 resources allocated by *iconv_open()*.19369 If a file descriptor is used to implement the type **iconv_t**, that file descriptor shall be closed.19370 **RETURN VALUE**19371 Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned and *errno* set to
19372 indicate the error.19373 **ERRORS**19374 The *iconv_close()* function may fail if:

19375 [EBADF] The conversion descriptor is invalid.

19376 **EXAMPLES**

19377 None.

19378 **APPLICATION USAGE**

19379 None.

19380 **RATIONALE**

19381 None.

19382 **FUTURE DIRECTIONS**

19383 None.

19384 **SEE ALSO**19385 *iconv()*, *iconv_open()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**iconv.h**>19386 **CHANGE HISTORY**

19387 First released in Issue 4. Derived from the HP-UX Manual.

19388 **NAME**

19389 iconv_open — codeset conversion allocation function

19390 **SYNOPSIS**19391 xSI `#include <iconv.h>`19392 `iconv_t iconv_open(const char *tocode, const char *fromcode);`

19393

19394 **DESCRIPTION**

19395 The *iconv_open()* function shall return a conversion descriptor that describes a conversion from
 19396 the codeset specified by the string pointed to by the *fromcode* argument to the codeset specified
 19397 by the string pointed to by the *tocode* argument. For state-dependent encodings, the conversion
 19398 descriptor shall be in a codeset-dependent initial shift state, ready for immediate use with
 19399 *iconv()*.

19400 Settings of *fromcode* and *tocode* and their permitted combinations are implementation-defined.19401 A conversion descriptor shall remain valid until it is closed by *iconv_close()* or an implicit close.

19402 If a file descriptor is used to implement conversion descriptors, the FD_CLOEXEC flag shall be
 19403 set; see <fcntl.h>.

19404 **RETURN VALUE**

19405 Upon successful completion, *iconv_open()* shall return a conversion descriptor for use on
 19406 subsequent calls to *iconv()*. Otherwise, *iconv_open()* shall return (**iconv_t**)-1 and set *errno* to
 19407 indicate the error.

19408 **ERRORS**19409 The *iconv_open()* function may fail if:

19410 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

19411 [ENFILE] Too many files are currently open in the system.

19412 [ENOMEM] Insufficient storage space is available.

19413 [EINVAL] The conversion specified by *fromcode* and *tocode* is not supported by the
 19414 implementation.

19415 **EXAMPLES**

19416 None.

19417 **APPLICATION USAGE**

19418 Some implementations of *iconv_open()* use *malloc()* to allocate space for internal buffer areas.
 19419 The *iconv_open()* function may fail if there is insufficient storage space to accommodate these
 19420 buffers.

19421 Conforming applications must assume that conversion descriptors are not valid after a call to
 19422 one of the *exec* functions.

19423 Application developers should consult the system documentation to determine the supported
 19424 codesets and their naming schemes.

19425 **RATIONALE**

19426 None.

19427 **FUTURE DIRECTIONS**

19428 None.

19429 **SEE ALSO**

19430 *iconv()*, *iconv_close()*, the Base Definitions volume of IEEE Std 1003.1-2001, <fcntl.h>, <iconv.h>

19431 **CHANGE HISTORY**

19432 First released in Issue 4. Derived from the HP-UX Manual.

19433 **NAME**

19434 if_freenameindex — free memory allocated by if_nameindex

19435 **SYNOPSIS**

19436 #include <net/if.h>

19437 void if_freenameindex(struct if_nameindex *ptr);

19438 **DESCRIPTION**

19439 The *if_freenameindex()* function shall free the memory allocated by *if_nameindex()*. The *ptr*
19440 argument shall be a pointer that was returned by *if_nameindex()*. After *if_freenameindex()* has
19441 been called, the application shall not use the array of which *ptr* is the address.

19442 **RETURN VALUE**

19443 None.

19444 **ERRORS**

19445 No errors are defined.

19446 **EXAMPLES**

19447 None.

19448 **APPLICATION USAGE**

19449 None.

19450 **RATIONALE**

19451 None.

19452 **FUTURE DIRECTIONS**

19453 None.

19454 **SEE ALSO**

19455 *getsockopt()*, *if_indextoname()*, *if_nameindex()*, *if_nametoindex()*, *setsockopt()*, the Base Definitions
19456 volume of IEEE Std 1003.1-2001, <net/if.h>

19457 **CHANGE HISTORY**

19458 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19459 **NAME**

19460 if_indextoname — map a network interface index to its corresponding name

19461 **SYNOPSIS**

19462 #include <net/if.h>

19463 char *if_indextoname(unsigned *ifindex*, char **ifname*);

19464 **DESCRIPTION**

19465 The *if_indextoname()* function shall map an interface index to its corresponding name.

19466 When this function is called, *ifname* shall point to a buffer of at least {IFNAMSIZ} bytes. The
19467 function shall place in this buffer the name of the interface with index *ifindex*.

19468 **RETURN VALUE**

19469 If *ifindex* is an interface index, then the function shall return the value supplied in *ifname*, which
19470 points to a buffer now containing the interface name. Otherwise, the function shall return a
19471 NULL pointer and set *errno* to indicate the error.

19472 **ERRORS**

19473 The *if_indextoname()* function shall fail if:

19474 [ENXIO] The interface does not exist.

19475 **EXAMPLES**

19476 None.

19477 **APPLICATION USAGE**

19478 None.

19479 **RATIONALE**

19480 None.

19481 **FUTURE DIRECTIONS**

19482 None.

19483 **SEE ALSO**

19484 *getsockopt()*, *if_freenameindex()*, *if_nameindex()*, *if_nametoindex()*, *setsockopt()*, the Base
19485 Definitions volume of IEEE Std 1003.1-2001, <net/if.h>

19486 **CHANGE HISTORY**

19487 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19488 **NAME**

19489 if_nameindex — return all network interface names and indexes

19490 **SYNOPSIS**

19491 #include <net/if.h>

19492 struct if_nameindex *if_nameindex(void);

19493 **DESCRIPTION**

19494 The *if_nameindex()* function shall return an array of *if_nameindex* structures, one structure per
19495 interface. The end of the array is indicated by a structure with an *if_index* field of zero and an
19496 *if_name* field of NULL.

19497 Applications should call *if_freenameindex()* to release the memory that may be dynamically
19498 allocated by this function, after they have finished using it.

19499 **RETURN VALUE**

19500 An array of structures identifying local interfaces. A NULL pointer is returned upon an error,
19501 with *errno* set to indicate the error.

19502 **ERRORS**

19503 The *if_nameindex()* function may fail if:

19504 [ENOBUFS] Insufficient resources are available to complete the function.

19505 **EXAMPLES**

19506 None.

19507 **APPLICATION USAGE**

19508 None.

19509 **RATIONALE**

19510 None.

19511 **FUTURE DIRECTIONS**

19512 None.

19513 **SEE ALSO**

19514 *getsockopt()*, *if_freenameindex()*, *if_indextoname()*, *if_nametoindex()*, *setsockopt()*, the Base
19515 Definitions volume of IEEE Std 1003.1-2001, <net/if.h>

19516 **CHANGE HISTORY**

19517 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19518 **NAME**

19519 if_nametoindex — map a network interface name to its corresponding index

19520 **SYNOPSIS**

19521 #include <net/if.h>

19522 unsigned if_nametoindex(const char *ifname);

19523 **DESCRIPTION**

19524 The *if_nametoindex()* function shall return the interface index corresponding to name *ifname*.

19525 **RETURN VALUE**

19526 The corresponding index if *ifname* is the name of an interface; otherwise, zero.

19527 **ERRORS**

19528 No errors are defined.

19529 **EXAMPLES**

19530 None.

19531 **APPLICATION USAGE**

19532 None.

19533 **RATIONALE**

19534 None.

19535 **FUTURE DIRECTIONS**

19536 None.

19537 **SEE ALSO**

19538 *getsockopt()*, *if_freenameindex()*, *if_indextoname()*, *if_nameindex()*, *setsockopt()*, the Base
19539 Definitions volume of IEEE Std 1003.1-2001, <net/if.h>

19540 **CHANGE HISTORY**

19541 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19542 **NAME**

19543 ilogb, ilogbf, ilogbl — return an unbiased exponent

19544 **SYNOPSIS**

19545 #include <math.h>

19546 int ilogb(double x);

19547 int ilogbf(float x);

19548 int ilogbl(long double x);

19549 **DESCRIPTION**

19550 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 19551 conflict between the requirements described here and the ISO C standard is unintentional. This
 19552 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

19553 These functions shall return the exponent part of their argument *x*. Formally, the return value is
 19554 the integral part of $\log_r |x|$ as a signed integral value, for non-zero *x*, where *r* is the radix of the
 19555 machine's floating-point arithmetic, which is the value of FLT_RADIX defined in <float.h>.

19556 An application wishing to check for error situations should set *errno* to zero and call
 19557 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 19558 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 19559 zero, an error has occurred.

19560 **RETURN VALUE**

19561 Upon successful completion, these functions shall return the exponent part of *x* as a signed
 19562 integer value. They are equivalent to calling the corresponding *logb*() function and casting the
 19563 returned value to type **int**.

19564 XSI If *x* is 0, a domain error shall occur, and the value FP_ILOGB0 shall be returned.

19565 XSI If *x* is $\pm\text{Inf}$, a domain error shall occur, and the value {INT_MAX} shall be returned.

19566 XSI If *x* is a NaN, a domain error shall occur, and the value FP_ILOGBNAN shall be returned.

19567 XSI If the correct value is greater than {INT_MAX}, {INT_MAX} shall be returned and a domain error
 19568 shall occur.

19569 If the correct value is less than {INT_MIN}, {INT_MIN} shall be returned and a domain error
 19570 shall occur.

19571 **ERRORS**

19572 These functions shall fail if:

19573 XSI	Domain Error	The <i>x</i> argument is zero, NaN, or $\pm\text{Inf}$, or the correct value is not representable as an integer.
-----------	--------------	---

19575	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then <i>errno</i> shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.
-------	--

19579 **EXAMPLES**

19580 None.

19581 **APPLICATION USAGE**

19582 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
19583 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

19584 **RATIONALE**

19585 The errors come from taking the expected floating-point value and converting it to **int**, which is
19586 an invalid operation in IEEE Std 754-1985 (since overflow, infinity, and NaN are not
19587 representable in a type **int**), so should be a domain error.

19588 There are no known implementations that overflow. For overflow to happen, {INT_MAX} must
19589 be less than $\text{LDBL_MAX_EXP} * \log_2(\text{FLT_RADIX})$ or {INT_MIN} must be greater than
19590 $\text{LDBL_MIN_EXP} * \log_2(\text{FLT_RADIX})$ if subnormals are not supported, or {INT_MIN} must be
19591 greater than $(\text{LDBL_MIN_EXP} - \text{LDBL_MANT_DIG}) * \log_2(\text{FLT_RADIX})$ if subnormals are
19592 supported.

19593 **FUTURE DIRECTIONS**

19594 None.

19595 **SEE ALSO**

19596 *feclearexcept()*, *fetestexcept()*, *logb()*, *scalb()*, the Base Definitions volume of IEEE Std 1003.1-2001,
19597 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <float.h>, <math.h>

19598 **CHANGE HISTORY**

19599 First released in Issue 4, Version 2.

19600 **Issue 5**

19601 Moved from X/OPEN UNIX extension to BASE.

19602 **Issue 6**19603 The *ilogb()* function is no longer marked as an extension.

19604 The *ilogbf()* and *ilogbl()* functions are added for alignment with the ISO/IEC 9899:1999
19605 standard.

19606 The RETURN VALUE section is revised for alignment with the ISO/IEC 9899:1999 standard.

19607 XSI extensions are marked.

19608 **NAME**

19609 imaxabs — return absolute value

19610 **SYNOPSIS**

19611 #include <inttypes.h>

19612 intmax_t imaxabs(intmax_t j);

19613 **DESCRIPTION**

19614 cx The functionality described on this reference page is aligned with the ISO C standard. Any
19615 conflict between the requirements described here and the ISO C standard is unintentional. This
19616 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

19617 The *imaxabs()* function shall compute the absolute value of an integer *j*. If the result cannot be
19618 represented, the behavior is undefined.

19619 **RETURN VALUE**19620 The *imaxabs()* function shall return the absolute value.19621 **ERRORS**

19622 No errors are defined.

19623 **EXAMPLES**

19624 None.

19625 **APPLICATION USAGE**

19626 The absolute value of the most negative number cannot be represented in two's complement.

19627 **RATIONALE**

19628 None.

19629 **FUTURE DIRECTIONS**

19630 None.

19631 **SEE ALSO**19632 *imaxdiv()*, the Base Definitions volume of IEEE Std 1003.1-2001, <inttypes.h>19633 **CHANGE HISTORY**

19634 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

19635 **NAME**19636 **imaxdiv** — return quotient and remainder19637 **SYNOPSIS**19638 `#include <inttypes.h>`19639 `imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);`19640 **DESCRIPTION**

19641 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
19642 conflict between the requirements described here and the ISO C standard is unintentional. This
19643 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

19644 The *imaxdiv()* function shall compute *numer* / *denom* and *numer* % *denom* in a single operation.19645 **RETURN VALUE**

19646 The *imaxdiv()* function shall return a structure of type **imaxdiv_t**, comprising both the quotient
19647 and the remainder. The structure shall contain (in either order) the members *quot* (the quotient)
19648 and *rem* (the remainder), each of which has type **intmax_t**.

19649 If either part of the result cannot be represented, the behavior is undefined.

19650 **ERRORS**

19651 No errors are defined.

19652 **EXAMPLES**

19653 None.

19654 **APPLICATION USAGE**

19655 None.

19656 **RATIONALE**

19657 None.

19658 **FUTURE DIRECTIONS**

19659 None.

19660 **SEE ALSO**19661 *imaxabs()*, the Base Definitions volume of IEEE Std 1003.1-2001, `<inttypes.h>`19662 **CHANGE HISTORY**

19663 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

19664 **NAME**19665 index — character string operations (**LEGACY**)19666 **SYNOPSIS**

19667 XSI #include <strings.h>

19668 char *index(const char *s, int c);

19669

19670 **DESCRIPTION**19671 The *index()* function shall be equivalent to *strchr()*.19672 **RETURN VALUE**19673 See *strchr()*.19674 **ERRORS**19675 See *strchr()*.19676 **EXAMPLES**

19677 None.

19678 **APPLICATION USAGE**19679 The *strchr()* function is preferred over this function.19680 For maximum portability, it is recommended to replace the function call to *index()* as follows:

19681 #define index(a,b) strchr((a),(b))

19682 **RATIONALE**

19683 None.

19684 **FUTURE DIRECTIONS**

19685 This function may be withdrawn in a future version.

19686 **SEE ALSO**19687 *strchr()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**strings.h**>19688 **CHANGE HISTORY**

19689 First released in Issue 4, Version 2.

19690 **Issue 5**

19691 Moved from X/OPEN UNIX extension to BASE.

19692 **Issue 6**

19693 This function is marked LEGACY.

19694 **NAME**

19695 inet_addr, inet_ntoa — IPv4 address manipulation

19696 **SYNOPSIS**

19697 #include <arpa/inet.h>

19698 in_addr_t inet_addr(const char *cp);

19699 char *inet_ntoa(struct in_addr in);

19700 **DESCRIPTION**19701 The *inet_addr()* function shall convert the string pointed to by *cp*, in the standard IPv4 dotted decimal notation, to an integer value suitable for use as an Internet address.19703 The *inet_ntoa()* function shall convert the Internet host address specified by *in* to a string in the Internet standard dot notation.19705 The *inet_ntoa()* function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

19707 All Internet addresses shall be returned in network order (bytes ordered from left to right).

19708 Values specified using IPv4 dotted decimal notation take one of the following forms:

19709 a.b.c.d When four parts are specified, each shall be interpreted as a byte of data and
19710 assigned, from left to right, to the four bytes of an Internet address.19711 a.b.c When a three-part address is specified, the last part shall be interpreted as a 16-bit
19712 quantity and placed in the rightmost two bytes of the network address. This makes
19713 the three-part address format convenient for specifying Class B network addresses
19714 as "128.net.host".19715 a.b When a two-part address is supplied, the last part shall be interpreted as a 24-bit
19716 quantity and placed in the rightmost three bytes of the network address. This
19717 makes the two-part address format convenient for specifying Class A network
19718 addresses as "net.host".19719 a When only one part is given, the value shall be stored directly in the network
19720 address without any byte rearrangement.19721 All numbers supplied as parts in IPv4 dotted decimal notation may be decimal, octal, or
19722 hexadecimal, as specified in the ISO C standard (that is, a leading 0x or 0X implies hexadecimal;
19723 otherwise, a leading '0' implies octal; otherwise, the number is interpreted as decimal).19724 **RETURN VALUE**19725 Upon successful completion, *inet_addr()* shall return the Internet address. Otherwise, it shall
19726 return (**in_addr_t**)(-1).19727 The *inet_ntoa()* function shall return a pointer to the network address in Internet standard dot
19728 notation.19729 **ERRORS**

19730 No errors are defined.

19731 EXAMPLES

19732 None.

19733 APPLICATION USAGE

19734 The return value of *inet_ntoa()* may point to static data that may be overwritten by subsequent
19735 calls to *inet_ntoa()*.

19736 RATIONALE

19737 None.

19738 FUTURE DIRECTIONS

19739 None.

19740 SEE ALSO

19741 *endhostent()*, *endnetent()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**arpa/inet.h**>

19742 CHANGE HISTORY

19743 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19744 **NAME**

19745 inet_ntop, inet_pton — convert IPv4 and IPv6 addresses between binary and text form

19746 **SYNOPSIS**

19747 #include <arpa/inet.h>

19748 const char *inet_ntop(int af, const void *restrict src,
19749 char *restrict dst, socklen_t size);

19750 int inet_pton(int af, const char *restrict src, void *restrict dst);

19751 **DESCRIPTION**

19752 The *inet_ntop()* function shall convert a numeric address into a text string suitable for
19753 IP6 presentation. The *af* argument shall specify the family of the address. This can be AF_INET or
19754 AF_INET6. The *src* argument points to a buffer holding an IPv4 address if the *af* argument is
19755 IP6 AF_INET, or an IPv6 address if the *af* argument is AF_INET6. The *dst* argument points to a
19756 buffer where the function stores the resulting text string; it shall not be NULL. The *size* argument
19757 specifies the size of this buffer, which shall be large enough to hold the text string
19758 IP6 (INET_ADDRSTRLEN characters for IPv4, INET6_ADDRSTRLEN characters for IPv6).

19759 The *inet_pton()* function shall convert an address in its standard text presentation form into its
19760 IP6 numeric binary form. The *af* argument shall specify the family of the address. The AF_INET and
19761 AF_INET6 address families shall be supported. The *src* argument points to the string being
19762 passed in. The *dst* argument points to a buffer into which the function stores the numeric
19763 IP6 address; this shall be large enough to hold the numeric address (32 bits for AF_INET, 128 bits for
19764 AF_INET6).

19765 If the *af* argument of *inet_pton()* is AF_INET, the *src* string shall be in the standard IPv4 dotted-
19766 decimal form:

19767 ddd.ddd.ddd.ddd

19768 where "ddd" is a one to three digit decimal number between 0 and 255 (see *inet_addr()*). The
19769 *inet_pton()* function does not accept other formats (such as the octal numbers, hexadecimal
19770 numbers, and fewer than four numbers that *inet_addr()* accepts).

19771 IP6 If the *af* argument of *inet_pton()* is AF_INET6, the *src* string shall be in one of the following
19772 standard IPv6 text forms:

19773 1. The preferred form is "x:x:x:x:x:x:x:x", where the 'x's are the hexadecimal values
19774 of the eight 16-bit pieces of the address. Leading zeros in individual fields can be omitted,
19775 but there shall be at least one numeral in every field.

19776 2. A string of contiguous zero fields in the preferred form can be shown as "::". The "::"
19777 can only appear once in an address. Unspecified addresses ("0:0:0:0:0:0:0:0") may
19778 be represented simply as "::".

19779 3. A third form that is sometimes more convenient when dealing with a mixed environment
19780 of IPv4 and IPv6 nodes is "x:x:x:x:x:x.d.d.d.d", where the 'x's are the
19781 hexadecimal values of the six high-order 16-bit pieces of the address, and the 'd's are the
19782 decimal values of the four low-order 8-bit pieces of the address (standard IPv4
19783 representation).

19784 **Note:** A more extensive description of the standard representations of IPv6 addresses can be found in
19785 RFC 2373.

19786

19787 **RETURN VALUE**

19788 The *inet_ntop()* function shall return a pointer to the buffer containing the text string if the
19789 conversion succeeds, and NULL otherwise, and set *errno* to indicate the error.

19790 The *inet_pton()* function shall return 1 if the conversion succeeds, with the address pointed to by
19791 *IP6* *dst* in network byte order. It shall return 0 if the input is not a valid IPv4 dotted-decimal string or
19792 a valid IPv6 address string, or -1 with *errno* set to [EAFNOSUPPORT] if the *af* argument is
19793 unknown.

19794 **ERRORS**

19795 The *inet_ntop()* and *inet_pton()* functions shall fail if:

19796 [EAFNOSUPPORT]

19797 The *af* argument is invalid.

19798 [ENOSPC] The size of the *inet_ntop()* result buffer is inadequate.

19799 **EXAMPLES**

19800 None.

19801 **APPLICATION USAGE**

19802 None.

19803 **RATIONALE**

19804 None.

19805 **FUTURE DIRECTIONS**

19806 None.

19807 **SEE ALSO**

19808 The Base Definitions volume of IEEE Std 1003.1-2001, <**arpa/inet.h**>

19809 **CHANGE HISTORY**

19810 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19811 IPv6 extensions are marked.

19812 The **restrict** keyword is added to the *inet_ntop()* and *inet_pton()* prototypes for alignment with
19813 the ISO/IEC 9899:1999 standard.

19814 **NAME**

19815 initstate, random, setstate, srandom — pseudo-random number functions

19816 **SYNOPSIS**

```

19817 xSI      #include <stdlib.h>

19818          char *initstate(unsigned seed, char *state, size_t size);
19819          long random(void);
19820          char *setstate(const char *state);
19821          void srandom(unsigned seed);
19822

```

19823 **DESCRIPTION**

19824 The *random()* function shall use a non-linear additive feedback random-number generator
 19825 employing a default state array size of 31 **long** integers to return successive pseudo-random
 19826 numbers in the range from 0 to $2^{31}-1$. The period of this random-number generator is
 19827 approximately $16 \times (2^{31}-1)$. The size of the state array determines the period of the random-
 19828 number generator. Increasing the state array size shall increase the period.

19829 With 256 bytes of state information, the period of the random-number generator shall be greater
 19830 than 2^{69} .

19831 Like *rand()*, *random()* shall produce by default a sequence of numbers that can be duplicated by
 19832 calling *srandom()* with 1 as the seed.

19833 The *srandom()* function shall initialize the current state array using the value of *seed*.

19834 The *initstate()* and *setstate()* functions handle restarting and changing random-number
 19835 generators. The *initstate()* function allows a state array, pointed to by the *state* argument, to be
 19836 initialized for future use. The *size* argument, which specifies the size in bytes of the state array,
 19837 shall be used by *initstate()* to decide what type of random-number generator to use; the larger
 19838 the state array, the more random the numbers. Values for the amount of state information are 8,
 19839 32, 64, 128, and 256 bytes. Other values greater than 8 bytes are rounded down to the nearest one
 19840 of these values. If *initstate()* is called with $8 \leq \text{size} < 32$, then *random()* shall use a simple linear
 19841 congruential random number generator. The *seed* argument specifies a starting point for the
 19842 random-number sequence and provides for restarting at the same point. The *initstate()* function
 19843 shall return a pointer to the previous state information array.

19844 If *initstate()* has not been called, then *random()* shall behave as though *initstate()* had been called
 19845 with *seed*=1 and *size*=128.

19846 Once a state has been initialized, *setstate()* allows switching between state arrays. The array
 19847 defined by the *state* argument shall be used for further random-number generation until
 19848 *initstate()* is called or *setstate()* is called again. The *setstate()* function shall return a pointer to the
 19849 previous state array.

19850 **RETURN VALUE**

19851 If *initstate()* is called with *size* less than 8, it shall return NULL.

19852 The *random()* function shall return the generated pseudo-random number.

19853 The *srandom()* function shall not return a value.

19854 Upon successful completion, *initstate()* and *setstate()* shall return a pointer to the previous state
 19855 array; otherwise, a null pointer shall be returned.

19856 **ERRORS**

19857 No errors are defined.

19858 **EXAMPLES**

19859 None.

19860 **APPLICATION USAGE**

19861 After initialization, a state array can be restarted at a different point in one of two ways:

- 19862 1. The *initstate()* function can be used, with the desired seed, state array, and size of the
19863 array.
- 19864 2. The *setstate()* function, with the desired state, can be used, followed by *srandom()* with the
19865 desired seed. The advantage of using both of these functions is that the size of the state
19866 array does not have to be saved once it is initialized.

19867 Although some implementations of *random()* have written messages to standard error, such
19868 implementations do not conform to IEEE Std 1003.1-2001.

19869 Issue 5 restored the historical behavior of this function.

19870 Threaded applications should use *rand_r()*, *erand48()*, *nrand48()*, or *jrand48()* instead of
19871 *random()* when an independent random number sequence in multiple threads is required.

19872 **RATIONALE**

19873 None.

19874 **FUTURE DIRECTIONS**

19875 None.

19876 **SEE ALSO**19877 *drand48()*, *rand()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>19878 **CHANGE HISTORY**

19879 First released in Issue 4, Version 2.

19880 **Issue 5**

19881 Moved from X/OPEN UNIX extension to BASE.

19882 In the DESCRIPTION, the phrase “values smaller than 8” is replaced with “values greater than
19883 or equal to 8, or less than 32”, “*size*<8” is replaced with “ $8 \leq \textit{size} < 32$ ”, and a new first paragraph
19884 is added to the RETURN VALUE section. A note is added to the APPLICATION USAGE
19885 indicating that these changes restore the historical behavior of the function.

19886 **Issue 6**

19887 In the DESCRIPTION, duplicate text “For values greater than or equal to 8 . . .” is removed.

19888 **NAME**

19889 insque, remque — insert or remove an element in a queue

19890 **SYNOPSIS**

```

19891 xSI      #include <search.h>

19892          void insque(void *element, void *pred);
19893          void remque(void *element);
19894

```

19895 **DESCRIPTION**

19896 The *insque()* and *remque()* functions shall manipulate queues built from doubly-linked lists. The
 19897 queue can be either circular or linear. An application using *insque()* or *remque()* shall ensure it
 19898 defines a structure in which the first two members of the structure are pointers to the same type
 19899 of structure, and any further members are application-specific. The first member of the structure
 19900 is a forward pointer to the next entry in the queue. The second member is a backward pointer to
 19901 the previous entry in the queue. If the queue is linear, the queue is terminated with null
 19902 pointers. The names of the structure and of the pointer members are not subject to any special
 19903 restriction.

19904 The *insque()* function shall insert the element pointed to by *element* into a queue immediately
 19905 after the element pointed to by *pred*.

19906 The *remque()* function shall remove the element pointed to by *element* from a queue.

19907 If the queue is to be used as a linear list, invoking *insque(&element, NULL)*, where *element* is the
 19908 initial element of the queue, shall initialize the forward and backward pointers of *element* to null
 19909 pointers.

19910 If the queue is to be used as a circular list, the application shall ensure it initializes the forward
 19911 pointer and the backward pointer of the initial element of the queue to the element's own
 19912 address.

19913 **RETURN VALUE**

19914 The *insque()* and *remque()* functions do not return a value.

19915 **ERRORS**

19916 No errors are defined.

19917 **EXAMPLES**19918 **Creating a Linear Linked List**

19919 The following example creates a linear linked list.

```

19920          #include <search.h>
19921          ...
19922          struct myque element1;
19923          struct myque element2;

19924          char *data1 = "DATA1";
19925          char *data2 = "DATA2";
19926          ...
19927          element1.data = data1;
19928          element2.data = data2;

19929          insque (&element1, NULL);
19930          insque (&element2, &element1);

```


Creating a Circular Linked List

The following example creates a circular linked list.

```
#include <search.h>
...
struct myque element1;
struct myque element2;

char *data1 = "DATA1";
char *data2 = "DATA2";
...
element1.data = data1;
element2.data = data2;

element1.fwd = &element1;
element1.bck = &element1;

insque (&element2, &element1);
```

Removing an Element

The following example removes the element pointed to by *element1*.

```
#include <search.h>
...
struct myque element1;
...
remque (&element1);
```

APPLICATION USAGE

The historical implementations of these functions described the arguments as being of type **struct qelem *** rather than as being of type **void *** as defined here. In those implementations, **struct qelem** was commonly defined in **<search.h>** as:

```
struct qelem {
    struct qelem  *q_forw;
    struct qelem  *q_back;
};
```

Applications using these functions, however, were never able to use this structure directly since it provided no room for the actual data contained in the elements. Most applications defined structures that contained the two pointers as the initial elements and also provided space for, or pointers to, the object's data. Applications that used these functions to update more than one type of table also had the problem of specifying two or more different structures with the same name, if they literally used **struct qelem** as specified.

As described here, the implementations were actually expecting a structure type where the first two members were forward and backward pointers to structures. With C compilers that didn't provide function prototypes, applications used structures as specified in the DESCRIPTION above and the compiler did what the application expected.

If this method had been carried forward with an ISO C standard compiler and the historical function prototype, most applications would have to be modified to cast pointers to the structures actually used to be pointers to **struct qelem** to avoid compilation warnings. By specifying **void *** as the argument type, applications do not need to change (unless they specifically referenced **struct qelem** and depended on it being defined in **<search.h>**).

19975 **RATIONALE**

19976 None.

19977 **FUTURE DIRECTIONS**

19978 None.

19979 **SEE ALSO**19980 The Base Definitions volume of IEEE Std 1003.1-2001, <**search.h**>19981 **CHANGE HISTORY**

19982 First released in Issue 4, Version 2.

19983 **Issue 5**

19984 Moved from X/OPEN UNIX extension to BASE.

19985 **Issue 6**

19986 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

19987 **NAME**19988 **ioctl** — control a STREAMS device (**STREAMS**)19989 **SYNOPSIS**19990 **XSR** #include <stropts.h>19991 int ioctl(int *fildev*, int *request*, ... /* *arg* */);

19992

19993 **DESCRIPTION**

19994 The *ioctl()* function shall perform a variety of control functions on STREAMS devices. For non-
 19995 STREAMS devices, the functions performed by this call are unspecified. The *request* argument
 19996 and an optional third argument (with varying type) shall be passed to and interpreted by the
 19997 appropriate part of the STREAM associated with *fildev*.

19998 The *fildev* argument is an open file descriptor that refers to a device.

19999 The *request* argument selects the control function to be performed and shall depend on the
 20000 STREAMS device being addressed.

20001 The *arg* argument represents additional information that is needed by this specific STREAMS
 20002 device to perform the requested function. The type of *arg* depends upon the particular control
 20003 request, but it shall be either an integer or a pointer to a device-specific data structure.

20004 The *ioctl()* commands applicable to STREAMS, their arguments, and error conditions that apply
 20005 to each individual command are described below.

20006 The following *ioctl()* commands, with error values indicated, are applicable to all STREAMS
 20007 files:

20008 **I_PUSH** Pushes the module whose name is pointed to by *arg* onto the top of the
 20009 current STREAM, just below the STREAM head. It then calls the *open()*
 20010 function of the newly-pushed module.

20011 The *ioctl()* function with the **I_PUSH** command shall fail if:

20012 [EINVAL] Invalid module name.

20013 [ENXIO] Open function of new module failed.

20014 [ENXIO] Hangup received on *fildev*.

20015 **I_POP** Removes the module just below the STREAM head of the STREAM pointed to
 20016 by *fildev*. The *arg* argument should be 0 in an **I_POP** request.

20017 The *ioctl()* function with the **I_POP** command shall fail if:

20018 [EINVAL] No module present in the STREAM.

20019 [ENXIO] Hangup received on *fildev*.

20020 **I_LOOK** Retrieves the name of the module just below the STREAM head of the
 20021 STREAM pointed to by *fildev*, and places it in a character string pointed to by
 20022 *arg*. The buffer pointed to by *arg* should be at least FMNAMESZ+1 bytes long,
 20023 where FMNAMESZ is defined in <stropts.h>.

20024 The *ioctl()* function with the **I_LOOK** command shall fail if:

20025 [EINVAL] No module present in the STREAM.

20026 **I_FLUSH** Flushes read and/or write queues, depending on the value of *arg*. Valid *arg*
 20027 values are:

20028	FLUSHR	Flush all read queues.
20029	FLUSHW	Flush all write queues.
20030	FLUSHRW	Flush all read and all write queues.
20031	The <i>ioctl()</i> function with the <i>I_FLUSH</i> command shall fail if:	
20032	[EINVAL]	Invalid <i>arg</i> value.
20033	[EAGAIN] or [ENOSR]	Unable to allocate buffers for flush message.
20034		
20035	[ENXIO]	Hangup received on <i>fildev</i> .
20036	I_FLUSHBAND	Flushes a particular band of messages. The <i>arg</i> argument points to a bandinfo structure. The <i>bi_flag</i> member may be one of FLUSHR, FLUSHW, or FLUSHRW as described above. The <i>bi_pri</i> member determines the priority band to be flushed.
20037		
20038		
20039		
20040	I_SETSIG	Requests that the STREAMS implementation send the SIGPOLL signal to the calling process when a particular event has occurred on the STREAM associated with <i>fildev</i> . I_SETSIG supports an asynchronous processing capability in STREAMS. The value of <i>arg</i> is a bitmask that specifies the events for which the process should be signaled. It is the bitwise-inclusive OR of any combination of the following constants:
20041		
20042		
20043		
20044		
20045		
20046	S_RDNORM	A normal (priority band set to 0) message has arrived at the head of a STREAM head read queue. A signal shall be generated even if the message is of zero length.
20047		
20048		
20049	S_RDBAND	A message with a non-zero priority band has arrived at the head of a STREAM head read queue. A signal shall be generated even if the message is of zero length.
20050		
20051		
20052	S_INPUT	A message, other than a high-priority message, has arrived at the head of a STREAM head read queue. A signal shall be generated even if the message is of zero length.
20053		
20054		
20055	S_HIPRI	A high-priority message is present on a STREAM head read queue. A signal shall be generated even if the message is of zero length.
20056		
20057		
20058	S_OUTPUT	The write queue for normal data (priority band 0) just below the STREAM head is no longer full. This notifies the process that there is room on the queue for sending (or writing) normal data downstream.
20059		
20060		
20061		
20062	S_WRNORM	Equivalent to S_OUTPUT.
20063	S_WRBAND	The write queue for a non-zero priority band just below the STREAM head is no longer full. This notifies the process that there is room on the queue for sending (or writing) priority data downstream.
20064		
20065		
20066		
20067	S_MSG	A STREAMS signal message that contains the SIGPOLL signal has reached the front of the STREAM head read queue.
20068		
20069		
20070	S_ERROR	Notification of an error condition has reached the STREAM head.
20071		

20072		S_HANGUP	Notification of a hangup has reached the STREAM head.
20073		S_BANDURG	When used in conjunction with S_RDBAND, SIGURG is generated instead of SIGPOLL when a priority message reaches the front of the STREAM head read queue.
20074			
20075			
20076			If <i>arg</i> is 0, the calling process shall be unregistered and shall not receive further SIGPOLL signals for the stream associated with <i>fildev</i> .
20077			
20078			Processes that wish to receive SIGPOLL signals shall ensure that they explicitly register to receive them using I_SETSIG. If several processes register to receive this signal for the same event on the same STREAM, each process shall be signaled when the event occurs.
20079			
20080			
20081			
20082			The <i>ioctl()</i> function with the I_SETSIG command shall fail if:
20083		[EINVAL]	The value of <i>arg</i> is invalid.
20084		[EINVAL]	The value of <i>arg</i> is 0 and the calling process is not registered to receive the SIGPOLL signal.
20085			
20086		[EAGAIN]	There were insufficient resources to store the signal request.
20087	I_GETSIG		Returns the events for which the calling process is currently registered to be sent a SIGPOLL signal. The events are returned as a bitmask in an <i>int</i> pointed to by <i>arg</i> , where the events are those specified in the description of I_SETSIG above.
20088			
20089			
20090			
20091			The <i>ioctl()</i> function with the I_GETSIG command shall fail if:
20092		[EINVAL]	Process is not registered to receive the SIGPOLL signal.
20093	I_FIND		Compares the names of all modules currently present in the STREAM to the name pointed to by <i>arg</i> , and returns 1 if the named module is present in the STREAM, or returns 0 if the named module is not present.
20094			
20095			
20096			The <i>ioctl()</i> function with the I_FIND command shall fail if:
20097		[EINVAL]	<i>arg</i> does not contain a valid module name.
20098	I_PEEK		Retrieves the information in the first message on the STREAM head read queue without taking the message off the queue. It is analogous to <i>getmsg()</i> except that this command does not remove the message from the queue. The <i>arg</i> argument points to a strpeek structure.
20099			
20100			
20101			
20102			The application shall ensure that the <i>maxlen</i> member in the ctlbuf and databuf strbuf structures is set to the number of bytes of control information and/or data information, respectively, to retrieve. The <i>flags</i> member may be marked RS_HIPRI or 0, as described by <i>getmsg()</i> . If the process sets <i>flags</i> to RS_HIPRI, for example, I_PEEK shall only look for a high-priority message on the STREAM head read queue.
20103			
20104			
20105			
20106			
20107			
20108			I_PEEK returns 1 if a message was retrieved, and returns 0 if no message was found on the STREAM head read queue, or if the RS_HIPRI flag was set in <i>flags</i> and a high-priority message was not present on the STREAM head read queue. It does not wait for a message to arrive. On return, ctlbuf specifies information in the control buffer, databuf specifies information in the data buffer, and <i>flags</i> contains the value RS_HIPRI or 0.
20109			
20110			
20111			
20112			
20113			
20114	I_SRDOPT		Sets the read mode using the value of the argument <i>arg</i> . Read modes are described in <i>read()</i> . Valid <i>arg</i> flags are:
20115			

20116	RNORM	Byte-stream mode, the default.
20117	RMSGD	Message-discard mode.
20118	RMSGN	Message-nondiscard mode.
20119	The bitwise-inclusive OR of RMSGD and RMSGN shall return [EINVAL]. The	
20120	bitwise-inclusive OR of RNORM and either RMSGD or RMSGN shall result in	
20121	the other flag overriding RNORM which is the default.	
20122	In addition, treatment of control messages by the STREAM head may be	
20123	changed by setting any of the following flags in <i>arg</i> :	
20124	RPROTNORM	Fail <i>read()</i> with [EBADMSG] if a message containing a
20125		control part is at the front of the STREAM head read queue.
20126	RPROTDAT	Deliver the control part of a message as data when a
20127		process issues a <i>read()</i> .
20128	RPROTDIS	Discard the control part of a message, delivering any data
20129		portion, when a process issues a <i>read()</i> .
20130	The <i>ioctl()</i> function with the I_SRDOPT command shall fail if:	
20131	[EINVAL]	The <i>arg</i> argument is not valid.
20132	I_GRDOPT	Returns the current read mode setting, as described above, in an int pointed to
20133		by the argument <i>arg</i> . Read modes are described in <i>read()</i> .
20134	I_NREAD	Counts the number of data bytes in the data part of the first message on the
20135		STREAM head read queue and places this value in the int pointed to by <i>arg</i> .
20136		The return value for the command shall be the number of messages on the
20137		STREAM head read queue. For example, if 0 is returned in <i>arg</i> , but the <i>ioctl()</i>
20138		return value is greater than 0, this indicates that a zero-length message is next
20139		on the queue.
20140	I_FDINSERT	Creates a message from specified buffer(s), adds information about another
20141		STREAM, and sends the message downstream. The message contains a
20142		control part and an optional data part. The data and control parts to be sent
20143		are distinguished by placement in separate buffers, as described below. The
20144		<i>arg</i> argument points to a strfdinsert structure.
20145	The application shall ensure that the <i>len</i> member in the ctlbuf strbuf structure	
20146	is set to the size of a t_uscalar_t plus the number of bytes of control	
20147	information to be sent with the message. The <i>fildev</i> member specifies the file	
20148	descriptor of the other STREAM, and the <i>offset</i> member, which must be	
20149	suitably aligned for use as a t_uscalar_t , specifies the offset from the start of	
20150	the control buffer where I_FDINSERT shall store a t_uscalar_t whose	
20151	interpretation is specific to the STREAM end. The application shall ensure that	
20152	the <i>len</i> member in the databuf strbuf structure is set to the number of bytes of	
20153	data information to be sent with the message, or to 0 if no data part is to be	
20154	sent.	
20155	The <i>flags</i> member specifies the type of message to be created. A normal	
20156	message is created if <i>flags</i> is set to 0, and a high-priority message is created if	
20157	<i>flags</i> is set to RS_HIPRI. For non-priority messages, I_FDINSERT shall block if	
20158	the STREAM write queue is full due to internal flow control conditions. For	
20159	priority messages, I_FDINSERT does not block on this condition. For non-	
20160	priority messages, I_FDINSERT does not block when the write queue is full	

20161		and O_NONBLOCK is set. Instead, it fails and sets <i>errno</i> to [EAGAIN].
20162		I_FDINSERT also blocks, unless prevented by lack of internal resources,
20163		waiting for the availability of message blocks in the STREAM, regardless of
20164		priority or whether O_NONBLOCK has been specified. No partial message is
20165		sent.
20166		The <i>ioctl()</i> function with the I_FDINSERT command shall fail if:
20167	[EAGAIN]	A non-priority message is specified, the O_NONBLOCK
20168		flag is set, and the STREAM write queue is full due to
20169		internal flow control conditions.
20170	[EAGAIN] or [ENOSR]	
20171		Buffers cannot be allocated for the message that is to be
20172		created.
20173	[EINVAL]	One of the following:
20174		— The <i>fildev</i> member of the strfdinsert structure is not a
20175		valid, open STREAM file descriptor.
20176		— The size of a t_uscalar_t plus <i>offset</i> is greater than the <i>len</i>
20177		member for the buffer specified through ctlbuf .
20178		— The <i>offset</i> member does not specify a properly-aligned
20179		location in the data buffer.
20180		— An undefined value is stored in <i>flags</i> .
20181	[ENXIO]	Hangup received on the STREAM identified by either the
20182		<i>fildev</i> argument or the <i>fildev</i> member of the strfdinsert
20183		structure.
20184	[ERANGE]	The <i>len</i> member for the buffer specified through databuf
20185		does not fall within the range specified by the maximum
20186		and minimum packet sizes of the topmost STREAM
20187		module; or the <i>len</i> member for the buffer specified through
20188		databuf is larger than the maximum configured size of the
20189		data part of a message; or the <i>len</i> member for the buffer
20190		specified through ctlbuf is larger than the maximum
20191		configured size of the control part of a message.
20192	I_STR	Constructs an internal STREAMS <i>ioctl()</i> message from the data pointed to by
20193		<i>arg</i> , and sends that message downstream.
20194		This mechanism is provided to send <i>ioctl()</i> requests to downstream modules
20195		and drivers. It allows information to be sent with <i>ioctl()</i> , and returns to the
20196		process any information sent upstream by the downstream recipient. I_STR
20197		shall block until the system responds with either a positive or negative
20198		acknowledgement message, or until the request times out after some period of
20199		time. If the request times out, it shall fail with <i>errno</i> set to [ETIME].
20200		At most, one I_STR can be active on a STREAM. Further I_STR calls shall
20201		block until the active I_STR completes at the STREAM head. The default
20202		timeout interval for these requests is 15 seconds. The O_NONBLOCK flag has
20203		no effect on this call.
20204		To send requests downstream, the application shall ensure that <i>arg</i> points to a
20205		strioc structure.

20206		The <i>ic_cmd</i> member is the internal <i>ioctl()</i> command intended for a downstream module or driver and <i>ic_timeout</i> is the number of seconds (−1=infinite, 0=use implementation-defined timeout interval, >0=as specified) an I_STR request shall wait for acknowledgement before timing out. <i>ic_len</i> is the number of bytes in the data argument, and <i>ic_dp</i> is a pointer to the data argument. The <i>ic_len</i> member has two uses: on input, it contains the length of the data argument passed in, and on return from the command, it contains the number of bytes being returned to the process (the buffer pointed to by <i>ic_dp</i> should be large enough to contain the maximum amount of data that any module or the driver in the STREAM can return).
20210		The STREAM head shall convert the information pointed to by the strioc structure to an internal <i>ioctl()</i> command message and send it downstream.
20211		The <i>ioctl()</i> function with the I_STR command shall fail if:
20212		[EAGAIN] or [ENOSR]
20213		Unable to allocate buffers for the <i>ioctl()</i> message.
20214		[EINVAL] The <i>ic_len</i> member is less than 0 or larger than the maximum configured size of the data part of a message, or <i>ic_timeout</i> is less than −1.
20215		[ENXIO] Hangup received on <i>fil</i> des.
20216		[ETIME] A downstream <i>ioctl()</i> timed out before acknowledgement was received.
20217		An I_STR can also fail while waiting for an acknowledgement if a message indicating an error or a hangup is received at the STREAM head. In addition, an error code can be returned in the positive or negative acknowledgement message, in the event the <i>ioctl()</i> command sent downstream fails. For these cases, I_STR shall fail with <i>errno</i> set to the value in the message.
20218	I_SWROPT	Sets the write mode using the value of the argument <i>arg</i> . Valid bit settings for <i>arg</i> are:
20219		SNDZERO Send a zero-length message downstream when a <i>write()</i> of 0 bytes occurs. To not send a zero-length message when a <i>write()</i> of 0 bytes occurs, the application shall ensure that this bit is not set in <i>arg</i> (for example, <i>arg</i> would be set to 0).
20220		The <i>ioctl()</i> function with the I_SWROPT command shall fail if:
20221		[EINVAL] <i>arg</i> is not the above value.
20222	I_GWROPT	Returns the current write mode setting, as described above, in the int that is pointed to by the argument <i>arg</i> .
20223	I_SENDFD	Creates a new reference to the open file description associated with the file descriptor <i>arg</i> , and writes a message on the STREAMS-based pipe <i>fil</i> des containing this reference, together with the user ID and group ID of the calling process.
20224		The <i>ioctl()</i> function with the I_SENDFD command shall fail if:
20225		[EAGAIN] The sending STREAM is unable to allocate a message block to contain the file pointer; or the read queue of the receiving STREAM head is full and cannot accept the message sent by I_SENDFD.

20251		[EBADF]	The <i>arg</i> argument is not a valid, open file descriptor.
20252		[EINVAL]	The <i>fildev</i> argument is not connected to a STREAM pipe.
20253		[ENXIO]	Hangup received on <i>fildev</i> .
20254	I_RECVFD		Retrieves the reference to an open file description from a message written to a STREAMS-based pipe using the I_SENDFD command, and allocates a new file descriptor in the calling process that refers to this open file description. The <i>arg</i> argument is a pointer to a strrecvfd data structure as defined in <stropts.h> .
20255			
20256			
20257			
20258			
20259			The <i>fd</i> member is a file descriptor. The <i>uid</i> and <i>gid</i> members are the effective user ID and effective group ID, respectively, of the sending process.
20260			
20261			If O_NONBLOCK is not set, I_RECVFD shall block until a message is present at the STREAM head. If O_NONBLOCK is set, I_RECVFD shall fail with <i>errno</i> set to [EAGAIN] if no message is present at the STREAM head.
20262			
20263			
20264			If the message at the STREAM head is a message sent by an I_SENDFD, a new file descriptor shall be allocated for the open file descriptor referenced in the message. The new file descriptor is placed in the <i>fd</i> member of the strrecvfd structure pointed to by <i>arg</i> .
20265			
20266			
20267			
20268			The <i>ioctl()</i> function with the I_RECVFD command shall fail if:
20269		[EAGAIN]	A message is not present at the STREAM head read queue and the O_NONBLOCK flag is set.
20270			
20271		[EBADMSG]	The message at the STREAM head read queue is not a message containing a passed file descriptor.
20272			
20273		[EMFILE]	The process has the maximum number of file descriptors currently open that it is allowed.
20274			
20275		[ENXIO]	Hangup received on <i>fildev</i> .
20276	I_LIST		Allows the process to list all the module names on the STREAM, up to and including the topmost driver name. If <i>arg</i> is a null pointer, the return value shall be the number of modules, including the driver, that are on the STREAM pointed to by <i>fildev</i> . This lets the process allocate enough space for the module names. Otherwise, it should point to a str_list structure.
20277			
20278			
20279			
20280			
20281			The <i>sl_nmods</i> member indicates the number of entries the process has allocated in the array. Upon return, the <i>sl_modlist</i> member of the str_list structure shall contain the list of module names, and the number of entries that have been filled into the <i>sl_modlist</i> array is found in the <i>sl_nmods</i> member (the number includes the number of modules including the driver). The return value from <i>ioctl()</i> shall be 0. The entries are filled in starting at the top of the STREAM and continuing downstream until either the end of the STREAM is reached, or the number of requested modules (<i>sl_nmods</i>) is satisfied.
20282			
20283			
20284			
20285			
20286			
20287			
20288			
20289			The <i>ioctl()</i> function with the I_LIST command shall fail if:
20290		[EINVAL]	The <i>sl_nmods</i> member is less than 1.
20291		[EAGAIN] or [ENOSR]	
20292			Unable to allocate buffers.
20293	I_ATMARK		Allows the process to see if the message at the head of the STREAM head read queue is marked by some module downstream. The <i>arg</i> argument determines
20294			

20295		how the checking is done when there may be multiple marked messages on
20296		the STREAM head read queue. It may take on the following values:
20297		ANYMARK Check if the message is marked.
20298		LASTMARK Check if the message is the last one marked on the queue.
20299		The bitwise-inclusive OR of the flags ANYMARK and LASTMARK is
20300		permitted.
20301		The return value shall be 1 if the mark condition is satisfied; otherwise, the
20302		value shall be 0.
20303		The <i>ioctl()</i> function with the I_ATMARK command shall fail if:
20304		[EINVAL] Invalid <i>arg</i> value.
20305	I_CKBAND	Checks if the message of a given priority band exists on the STREAM head
20306		read queue. This shall return 1 if a message of the given priority exists, 0 if no
20307		such message exists, or -1 on error. <i>arg</i> should be of type int .
20308		The <i>ioctl()</i> function with the I_CKBAND command shall fail if:
20309		[EINVAL] Invalid <i>arg</i> value.
20310	I_GETBAND	Returns the priority band of the first message on the STREAM head read
20311		queue in the integer referenced by <i>arg</i> .
20312		The <i>ioctl()</i> function with the I_GETBAND command shall fail if:
20313		[ENODATA] No message on the STREAM head read queue.
20314	I_CANPUT	Checks if a certain band is writable. <i>arg</i> is set to the priority band in question.
20315		The return value shall be 0 if the band is flow-controlled, 1 if the band is
20316		writable, or -1 on error.
20317		The <i>ioctl()</i> function with the I_CANPUT command shall fail if:
20318		[EINVAL] Invalid <i>arg</i> value.
20319	I_SETCLTIME	This request allows the process to set the time the STREAM head shall delay
20320		when a STREAM is closing and there is data on the write queues. Before
20321		closing each module or driver, if there is data on its write queue, the STREAM
20322		head shall delay for the specified amount of time to allow the data to drain. If,
20323		after the delay, data is still present, it shall be flushed. The <i>arg</i> argument is a
20324		pointer to an integer specifying the number of milliseconds to delay, rounded
20325		up to the nearest valid value. If I_SETCLTIME is not performed on a STREAM,
20326		an implementation-defined default timeout interval is used.
20327		The <i>ioctl()</i> function with the I_SETCLTIME command shall fail if:
20328		[EINVAL] Invalid <i>arg</i> value.
20329	I_GETCLTIME	Returns the close time delay in the integer pointed to by <i>arg</i> .

20330 **Multiplexed STREAMS Configurations**

20331 The following commands are used for connecting and disconnecting multiplexed STREAMS
20332 configurations. These commands use an implementation-defined default timeout interval.

20333 **I_LINK** Connects two STREAMs, where *fildev* is the file descriptor of the STREAM
20334 connected to the multiplexing driver, and *arg* is the file descriptor of the
20335 STREAM connected to another driver. The STREAM designated by *arg* is
20336 connected below the multiplexing driver. I_LINK requires the multiplexing
20337 driver to send an acknowledgement message to the STREAM head regarding
20338 the connection. This call shall return a multiplexer ID number (an identifier
20339 used to disconnect the multiplexer; see I_UNLINK) on success, and -1 on
20340 failure.

20341 The *ioctl()* function with the I_LINK command shall fail if:

- 20342 [ENXIO] Hangup received on *fildev*.
- 20343 [ETIME] Timeout before acknowledgement message was received at
20344 STREAM head.
- 20345 [EAGAIN] or [ENOSR]
20346 Unable to allocate STREAMS storage to perform the
20347 I_LINK.
- 20348 [EBADF] The *arg* argument is not a valid, open file descriptor.
- 20349 [EINVAL] The *fildev* argument does not support multiplexing; or *arg* is
20350 not a STREAM or is already connected downstream from a
20351 multiplexer; or the specified I_LINK operation would
20352 connect the STREAM head in more than one place in the
20353 multiplexed STREAM.

20354 An I_LINK can also fail while waiting for the multiplexing driver to
20355 acknowledge the request, if a message indicating an error or a hangup is
20356 received at the STREAM head of *fildev*. In addition, an error code can be
20357 returned in the positive or negative acknowledgement message. For these
20358 cases, I_LINK fails with *errno* set to the value in the message.

20359 **I_UNLINK** Disconnects the two STREAMs specified by *fildev* and *arg*. *fildev* is the file
20360 descriptor of the STREAM connected to the multiplexing driver. The *arg*
20361 argument is the multiplexer ID number that was returned by the I_LINK
20362 *ioctl()* command when a STREAM was connected downstream from the
20363 multiplexing driver. If *arg* is MUXID_ALL, then all STREAMs that were
20364 connected to *fildev* shall be disconnected. As in I_LINK, this command
20365 requires acknowledgement.

20366 The *ioctl()* function with the I_UNLINK command shall fail if:

- 20367 [ENXIO] Hangup received on *fildev*.
- 20368 [ETIME] Timeout before acknowledgement message was received at
20369 STREAM head.
- 20370 [EAGAIN] or [ENOSR]
20371 Unable to allocate buffers for the acknowledgement
20372 message.
- 20373 [EINVAL] Invalid multiplexer ID number.

20374		An I_UNLINK can also fail while waiting for the multiplexing driver to
20375		acknowledge the request if a message indicating an error or a hangup is
20376		received at the STREAM head of <i>filde</i> s. In addition, an error code can be
20377		returned in the positive or negative acknowledgement message. For these
20378		cases, I_UNLINK shall fail with <i>errno</i> set to the value in the message.
20379	I_PLINK	Creates a <i>persistent connection</i> between two STREAMs, where <i>filde</i> s is the file
20380		descriptor of the STREAM connected to the multiplexing driver, and <i>arg</i> is the
20381		file descriptor of the STREAM connected to another driver. This call shall
20382		create a persistent connection which can exist even if the file descriptor <i>filde</i> s
20383		associated with the upper STREAM to the multiplexing driver is closed. The
20384		STREAM designated by <i>arg</i> gets connected via a persistent connection below
20385		the multiplexing driver. I_PLINK requires the multiplexing driver to send an
20386		acknowledgement message to the STREAM head. This call shall return a
20387		multiplexer ID number (an identifier that may be used to disconnect the
20388		multiplexer; see I_PUNLINK) on success, and -1 on failure.
20389		The <i>ioctl</i> () function with the I_PLINK command shall fail if:
20390	[ENXIO]	Hangup received on <i>filde</i> s.
20391	[ETIME]	Timeout before acknowledgement message was received at
20392		STREAM head.
20393	[EAGAIN] or [ENOSR]	
20394		Unable to allocate STREAMS storage to perform the
20395		I_PLINK.
20396	[EBADF]	The <i>arg</i> argument is not a valid, open file descriptor.
20397	[EINVAL]	The <i>filde</i> s argument does not support multiplexing; or <i>arg</i> is
20398		not a STREAM or is already connected downstream from a
20399		multiplexer; or the specified I_PLINK operation would
20400		connect the STREAM head in more than one place in the
20401		multiplexed STREAM.
20402		An I_PLINK can also fail while waiting for the multiplexing driver to
20403		acknowledge the request, if a message indicating an error or a hangup is
20404		received at the STREAM head of <i>filde</i> s. In addition, an error code can be
20405		returned in the positive or negative acknowledgement message. For these
20406		cases, I_PLINK shall fail with <i>errno</i> set to the value in the message.
20407	I_PUNLINK	Disconnects the two STREAMs specified by <i>filde</i> s and <i>arg</i> from a persistent
20408		connection. The <i>filde</i> s argument is the file descriptor of the STREAM
20409		connected to the multiplexing driver. The <i>arg</i> argument is the multiplexer ID
20410		number that was returned by the I_PLINK <i>ioctl</i> () command when a STREAM
20411		was connected downstream from the multiplexing driver. If <i>arg</i> is
20412		MUXID_ALL, then all STREAMs which are persistent connections to <i>filde</i> s
20413		shall be disconnected. As in I_PLINK, this command requires the multiplexing
20414		driver to acknowledge the request.
20415		The <i>ioctl</i> () function with the I_PUNLINK command shall fail if:
20416	[ENXIO]	Hangup received on <i>filde</i> s.
20417	[ETIME]	Timeout before acknowledgement message was received at
20418		STREAM head.

20419 [EAGAIN] or [ENOSR]
 20420 Unable to allocate buffers for the acknowledgement
 20421 message.
 20422 [EINVAL] Invalid multiplexer ID number.
 20423 An I_PUNLINK can also fail while waiting for the multiplexing driver to
 20424 acknowledge the request if a message indicating an error or a hangup is
 20425 received at the STREAM head of *fildev*. In addition, an error code can be
 20426 returned in the positive or negative acknowledgement message. For these
 20427 cases, I_PUNLINK shall fail with *errno* set to the value in the message.

20428 RETURN VALUE

20429 Upon successful completion, *ioctl()* shall return a value other than -1 that depends upon the
 20430 STREAMS device control function. Otherwise, it shall return -1 and set *errno* to indicate the
 20431 error.

20432 ERRORS

20433 Under the following general conditions, *ioctl()* shall fail if:

20434 [EBADF] The *fildev* argument is not a valid open file descriptor.
 20435 [EINTR] A signal was caught during the *ioctl()* operation.
 20436 [EINVAL] The STREAM or multiplexer referenced by *fildev* is linked (directly or
 20437 indirectly) downstream from a multiplexer.

20438 If an underlying device driver detects an error, then *ioctl()* shall fail if:

20439 [EINVAL] The *request* or *arg* argument is not valid for this device.
 20440 [EIO] Some physical I/O error has occurred.
 20441 [ENOTTY] The *fildev* argument is not associated with a STREAMS device that accepts
 20442 control functions.
 20443 [ENXIO] The *request* and *arg* arguments are valid for this device driver, but the service
 20444 requested cannot be performed on this particular sub-device.
 20445 [ENODEV] The *fildev* argument refers to a valid STREAMS device, but the corresponding
 20446 device driver does not support the *ioctl()* function.

20447 If a STREAM is connected downstream from a multiplexer, any *ioctl()* command except
 20448 I_UNLINK and I_PUNLINK shall set *errno* to [EINVAL].

20449 EXAMPLES

20450 None.

20451 APPLICATION USAGE

20452 The implementation-defined timeout interval for STREAMS has historically been 15 seconds.

20453 RATIONALE

20454 None.

20455 FUTURE DIRECTIONS

20456 None.

20457 SEE ALSO

20458 Section 2.6 (on page 38), *close()*, *fcntl()*, *getmsg()*, *open()*, *pipe()*, *poll()*, *putmsg()*, *read()*,
 20459 *sigaction()*, *write()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**stropts.h**>

20460 CHANGE HISTORY

20461 First released in Issue 4, Version 2.

20462 Issue 5

20463 Moved from X/OPEN UNIX extension to BASE.

20464 Issue 6

20465 The Open Group Corrigendum U028/4 is applied, correcting text in the L_FDINSERT [EINVAL]
20466 case to refer to *ctlbuf*.

20467 This function is marked as part of the XSI STREAMS Option Group.

20468 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

20469 **NAME**

20470 isalnum — test for an alphanumeric character

20471 **SYNOPSIS**

20472 #include <ctype.h>

20473 int isalnum(int c);

20474 **DESCRIPTION**

20475 cx The functionality described on this reference page is aligned with the ISO C standard. Any
20476 conflict between the requirements described here and the ISO C standard is unintentional. This
20477 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

20478 The *isalnum()* function shall test whether *c* is a character of class **alpha** or **digit** in the program's
20479 current locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

20480 The *c* argument is an **int**, the value of which the application shall ensure is representable as an
20481 **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the
20482 behavior is undefined.

20483 **RETURN VALUE**

20484 The *isalnum()* function shall return non-zero if *c* is an alphanumeric character; otherwise, it shall
20485 return 0.

20486 **ERRORS**

20487 No errors are defined.

20488 **EXAMPLES**

20489 None.

20490 **APPLICATION USAGE**

20491 To ensure applications portability, especially across natural languages, only this function and
20492 those listed in the SEE ALSO section should be used for character classification.

20493 **RATIONALE**

20494 None.

20495 **FUTURE DIRECTIONS**

20496 None.

20497 **SEE ALSO**

20498 *isalpha()*, *isctrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*, *isxdigit()*,
20499 *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <ctype.h>,
20500 <stdio.h>

20501 **CHANGE HISTORY**

20502 First released in Issue 1. Derived from Issue 1 of the SVID.

20503 **Issue 6**

20504 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

20505 **NAME**

20506 isalpha — test for an alphabetic character

20507 **SYNOPSIS**

20508 #include <ctype.h>

20509 int isalpha(int c);

20510 **DESCRIPTION**

20511 cx The functionality described on this reference page is aligned with the ISO C standard. Any
20512 conflict between the requirements described here and the ISO C standard is unintentional. This
20513 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

20514 The *isalpha()* function shall test whether *c* is a character of class **alpha** in the program's current
20515 locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

20516 The *c* argument is an **int**, the value of which the application shall ensure is representable as an
20517 **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the
20518 behavior is undefined.

20519 **RETURN VALUE**

20520 The *isalpha()* function shall return non-zero if *c* is an alphabetic character; otherwise, it shall
20521 return 0.

20522 **ERRORS**

20523 No errors are defined.

20524 **EXAMPLES**

20525 None.

20526 **APPLICATION USAGE**

20527 To ensure applications portability, especially across natural languages, only this function and
20528 those listed in the SEE ALSO section should be used for character classification.

20529 **RATIONALE**

20530 None.

20531 **FUTURE DIRECTIONS**

20532 None.

20533 **SEE ALSO**

20534 *isalnum()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*,
20535 *isxdigit()*, *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale,
20536 <ctype.h>, <stdio.h>

20537 **CHANGE HISTORY**

20538 First released in Issue 1. Derived from Issue 1 of the SVID.

20539 **Issue 6**

20540 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

20541 **NAME**

20542 isascii — test for a 7-bit US-ASCII character

20543 **SYNOPSIS**

20544 xSI #include <ctype.h>

20545 int isascii(int c);

20546

20547 **DESCRIPTION**20548 The *isascii()* function shall test whether *c* is a 7-bit US-ASCII character code.20549 The *isascii()* function is defined on all integer values.20550 **RETURN VALUE**20551 The *isascii()* function shall return non-zero if *c* is a 7-bit US-ASCII character code between 0 and
20552 octal 0177 inclusive; otherwise, it shall return 0.20553 **ERRORS**

20554 No errors are defined.

20555 **EXAMPLES**

20556 None.

20557 **APPLICATION USAGE**

20558 None.

20559 **RATIONALE**

20560 None.

20561 **FUTURE DIRECTIONS**

20562 None.

20563 **SEE ALSO**

20564 The Base Definitions volume of IEEE Std 1003.1-2001, <ctype.h>

20565 **CHANGE HISTORY**

20566 First released in Issue 1. Derived from Issue 1 of the SVID.

20567 **NAME**20568 isastream — test a file descriptor (**STREAMS**)20569 **SYNOPSIS**

20570 xSR #include <stropts.h>

20571 int isastream(int *fildev*);

20572

20573 **DESCRIPTION**20574 The *isastream()* function shall test whether *fildev*, an open file descriptor, is associated with a
20575 STREAMS-based file.20576 **RETURN VALUE**20577 Upon successful completion, *isastream()* shall return 1 if *fildev* refers to a STREAMS-based file
20578 and 0 if not. Otherwise, *isastream()* shall return -1 and set *errno* to indicate the error.20579 **ERRORS**20580 The *isastream()* function shall fail if:20581 [EBADF] The *fildev* argument is not a valid open file descriptor.20582 **EXAMPLES**

20583 None.

20584 **APPLICATION USAGE**

20585 None.

20586 **RATIONALE**

20587 None.

20588 **FUTURE DIRECTIONS**

20589 None.

20590 **SEE ALSO**

20591 The Base Definitions volume of IEEE Std 1003.1-2001, <stropts.h>

20592 **CHANGE HISTORY**

20593 First released in Issue 4, Version 2.

20594 **Issue 5**

20595 Moved from X/OPEN UNIX extension to BASE.

20596 **NAME**

20597 isatty — test for a terminal device

20598 **SYNOPSIS**

20599 #include <unistd.h>

20600 int isatty(int *fdes*);

20601 **DESCRIPTION**

20602 The *isatty()* function shall test whether *fdes*, an open file descriptor, is associated with a
20603 terminal device.

20604 **RETURN VALUE**

20605 The *isatty()* function shall return 1 if *fdes* is associated with a terminal; otherwise, it shall return
20606 0 and may set *errno* to indicate the error.

20607 **ERRORS**

20608 The *isatty()* function may fail if:

20609 [EBADF] The *fdes* argument is not a valid open file descriptor.

20610 [ENOTTY] The *fdes* argument is not associated with a terminal.

20611 **EXAMPLES**

20612 None.

20613 **APPLICATION USAGE**

20614 The *isatty()* function does not necessarily indicate that a human being is available for interaction
20615 via *fdes*. It is quite possible that non-terminal devices are connected to the communications
20616 line.

20617 **RATIONALE**

20618 None.

20619 **FUTURE DIRECTIONS**

20620 None.

20621 **SEE ALSO**

20622 The Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

20623 **CHANGE HISTORY**

20624 First released in Issue 1. Derived from Issue 1 of the SVID.

20625 **Issue 6**

20626 The following new requirements on POSIX implementations derive from alignment with the
20627 Single UNIX Specification:

- 20628 • The optional setting of *errno* to indicate an error is added.
- 20629 • The [EBADF] and [ENOTTY] optional error conditions are added.

20630 **NAME**

20631 isblank — test for a blank character

20632 **SYNOPSIS**

20633 #include <ctype.h>

20634 int isblank(int c);

20635 **DESCRIPTION**

20636 cx The functionality described on this reference page is aligned with the ISO C standard. Any
20637 conflict between the requirements described here and the ISO C standard is unintentional. This
20638 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

20639 The *isblank()* function shall test whether *c* is a character of class **blank** in the program's current
20640 locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

20641 The *c* argument is a type **int**, the value of which the application shall ensure is a character
20642 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
20643 any other value, the behavior is undefined.

20644 **RETURN VALUE**20645 The *isblank()* function shall return non-zero if *c* is a <blank>; otherwise, it shall return 0.20646 **ERRORS**

20647 No errors are defined.

20648 **EXAMPLES**

20649 None.

20650 **APPLICATION USAGE**

20651 To ensure applications portability, especially across natural languages, only this function and
20652 those listed in the SEE ALSO section should be used for character classification.

20653 **RATIONALE**

20654 None.

20655 **FUTURE DIRECTIONS**

20656 None.

20657 **SEE ALSO**

20658 *isalnum()*, *isalpha()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*,
20659 *isxdigit()*, *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale,
20660 <ctype.h>

20661 **CHANGE HISTORY**

20662 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

20663 **NAME**

20664 isctrl — test for a control character

20665 **SYNOPSIS**

20666 #include <ctype.h>

20667 int isctrl(int c);

20668 **DESCRIPTION**

20669 cx The functionality described on this reference page is aligned with the ISO C standard. Any
20670 conflict between the requirements described here and the ISO C standard is unintentional. This
20671 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

20672 The *isctrl()* function shall test whether *c* is a character of class **cntrl** in the program's current
20673 locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

20674 The *c* argument is a type **int**, the value of which the application shall ensure is a character
20675 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
20676 any other value, the behavior is undefined.

20677 **RETURN VALUE**20678 The *isctrl()* function shall return non-zero if *c* is a control character; otherwise, it shall return 0.20679 **ERRORS**

20680 No errors are defined.

20681 **EXAMPLES**

20682 None.

20683 **APPLICATION USAGE**

20684 To ensure applications portability, especially across natural languages, only this function and
20685 those listed in the SEE ALSO section should be used for character classification.

20686 **RATIONALE**

20687 None.

20688 **FUTURE DIRECTIONS**

20689 None.

20690 **SEE ALSO**

20691 *isalnum()*, *isalpha()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*,
20692 *isxdigit()*, *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale,
20693 <ctype.h>

20694 **CHANGE HISTORY**

20695 First released in Issue 1. Derived from Issue 1 of the SVID.

20696 **Issue 6**

20697 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

20698 **NAME**20699 `isdigit` — test for a decimal digit20700 **SYNOPSIS**20701 `#include <ctype.h>`20702 `int isdigit(int c);`20703 **DESCRIPTION**

20704 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
20705 conflict between the requirements described here and the ISO C standard is unintentional. This
20706 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

20707 The *isdigit()* function shall test whether *c* is a character of class **digit** in the program's current
20708 locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

20709 The *c* argument is an **int**, the value of which the application shall ensure is a character
20710 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
20711 any other value, the behavior is undefined.

20712 **RETURN VALUE**20713 The *isdigit()* function shall return non-zero if *c* is a decimal digit; otherwise, it shall return 0.20714 **ERRORS**

20715 No errors are defined.

20716 **EXAMPLES**

20717 None.

20718 **APPLICATION USAGE**

20719 To ensure applications portability, especially across natural languages, only this function and
20720 those listed in the SEE ALSO section should be used for character classification.

20721 **RATIONALE**

20722 None.

20723 **FUTURE DIRECTIONS**

20724 None.

20725 **SEE ALSO**

20726 *isalnum()*, *isalpha()*, *iscntrl()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*,
20727 *isxdigit()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, `<ctype.h>`

20728 **CHANGE HISTORY**

20729 First released in Issue 1. Derived from Issue 1 of the SVID.

20730 **Issue 6**

20731 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

20732 **NAME**

20733 isfinite — test for finite value

20734 **SYNOPSIS**

20735 #include <math.h>

20736 int isfinite(real-floating x);

20737 **DESCRIPTION**

20738 cx The functionality described on this reference page is aligned with the ISO C standard. Any
20739 conflict between the requirements described here and the ISO C standard is unintentional. This
20740 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

20741 The *isfinite()* macro shall determine whether its argument has a finite value (zero, subnormal, or
20742 normal, and not infinite or NaN). First, an argument represented in a format wider than its
20743 semantic type is converted to its semantic type. Then determination is based on the type of the
20744 argument.

20745 **RETURN VALUE**20746 The *isfinite()* macro shall return a non-zero value if and only if its argument has a finite value.20747 **ERRORS**

20748 No errors are defined.

20749 **EXAMPLES**

20750 None.

20751 **APPLICATION USAGE**

20752 None.

20753 **RATIONALE**

20754 None.

20755 **FUTURE DIRECTIONS**

20756 None.

20757 **SEE ALSO**

20758 *fpclassify()*, *isinf()*, *isnan()*, *isnormal()*, *signbit()*, the Base Definitions volume of
20759 IEEE Std 1003.1-2001 <math.h>

20760 **CHANGE HISTORY**

20761 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

20762 **NAME**

20763 isgraph — test for a visible character

20764 **SYNOPSIS**

20765 #include <ctype.h>

20766 int isgraph(int c);

20767 **DESCRIPTION**

20768 cx The functionality described on this reference page is aligned with the ISO C standard. Any
20769 conflict between the requirements described here and the ISO C standard is unintentional. This
20770 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

20771 The *isgraph()* function shall test whether *c* is a character of class **graph** in the program's current
20772 locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

20773 The *c* argument is an **int**, the value of which the application shall ensure is a character
20774 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
20775 any other value, the behavior is undefined.

20776 **RETURN VALUE**

20777 The *isgraph()* function shall return non-zero if *c* is a character with a visible representation;
20778 otherwise, it shall return 0.

20779 **ERRORS**

20780 No errors are defined.

20781 **EXAMPLES**

20782 None.

20783 **APPLICATION USAGE**

20784 To ensure applications portability, especially across natural languages, only this function and
20785 those listed in the SEE ALSO section should be used for character classification.

20786 **RATIONALE**

20787 None.

20788 **FUTURE DIRECTIONS**

20789 None.

20790 **SEE ALSO**

20791 *isalnum()*, *isalpha()*, *iscntrl()*, *isdigit()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*, *isxdigit()*,
20792 *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <ctype.h>

20793 **CHANGE HISTORY**

20794 First released in Issue 1. Derived from Issue 1 of the SVID.

20795 **Issue 6**

20796 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

20797 **NAME**

20798 isgreater — test if x greater than y

20799 **SYNOPSIS**

20800 #include <math.h>

20801 int isgreater(real-floating x, real-floating y);

20802 **DESCRIPTION**

20803 cx The functionality described on this reference page is aligned with the ISO C standard. Any
20804 conflict between the requirements described here and the ISO C standard is unintentional. This
20805 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

20806 The *isgreater()* macro shall determine whether its first argument is greater than its second
20807 argument. The value of *isgreater(x, y)* shall be equal to $(x) > (y)$; however, unlike $(x) > (y)$,
20808 *isgreater(x, y)* shall not raise the invalid floating-point exception when x and y are unordered.

20809 **RETURN VALUE**20810 Upon successful completion, the *isgreater()* macro shall return the value of $(x) > (y)$.

20811 If x or y is NaN, 0 shall be returned.

20812 **ERRORS**

20813 No errors are defined.

20814 **EXAMPLES**

20815 None.

20816 **APPLICATION USAGE**

20817 The relational and equality operators support the usual mathematical relationships between
20818 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
20819 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
20820 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
20821 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
20822 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
20823 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
20824 indicates that the argument shall be an expression of **real-floating** type.

20825 **RATIONALE**

20826 None.

20827 **FUTURE DIRECTIONS**

20828 None.

20829 **SEE ALSO**

20830 *isgreaterequal()*, *isless()*, *islessequal()*, *islessgreater()*, *isunordered()*, the Base Definitions volume of
20831 IEEE Std 1003.1-2001 <math.h>

20832 **CHANGE HISTORY**

20833 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

20834 **NAME**

20835 isgreaterequal — test if *x* is greater than or equal to *y*

20836 **SYNOPSIS**

20837 #include <math.h>

20838 int isgreaterequal(real-floating *x*, real-floating *y*);

20839 **DESCRIPTION**

20840 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
20841 conflict between the requirements described here and the ISO C standard is unintentional. This
20842 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

20843 The *isgreaterequal()* macro shall determine whether its first argument is greater than or equal to
20844 its second argument. The value of *isgreaterequal*(*x*, *y*) shall be equal to (*x*) >= (*y*); however, unlike
20845 (*x*) >= (*y*), *isgreaterequal*(*x*, *y*) shall not raise the invalid floating-point exception when *x* and *y* are
20846 unordered.

20847 **RETURN VALUE**

20848 Upon successful completion, the *isgreaterequal()* macro shall return the value of (*x*) >= (*y*).

20849 If *x* or *y* is NaN, 0 shall be returned.

20850 **ERRORS**

20851 No errors are defined.

20852 **EXAMPLES**

20853 None.

20854 **APPLICATION USAGE**

20855 The relational and equality operators support the usual mathematical relationships between
20856 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
20857 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
20858 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
20859 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
20860 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
20861 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
20862 indicates that the argument shall be an expression of **real-floating** type.

20863 **RATIONALE**

20864 None.

20865 **FUTURE DIRECTIONS**

20866 None.

20867 **SEE ALSO**

20868 *isgreater()*, *isless()*, *islessequal()*, *islessgreater()*, *isunordered()*, the Base Definitions volume of
20869 IEEE Std 1003.1-2001 <**math.h**>

20870 **CHANGE HISTORY**

20871 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

20872 **NAME**

20873 isinf — test for infinity

20874 **SYNOPSIS**

20875 #include <math.h>

20876 int isinf(real-floating x);

20877 **DESCRIPTION**

20878 cx The functionality described on this reference page is aligned with the ISO C standard. Any
20879 conflict between the requirements described here and the ISO C standard is unintentional. This
20880 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

20881 The *isinf()* macro shall determine whether its argument value is an infinity (positive or
20882 negative). First, an argument represented in a format wider than its semantic type is converted
20883 to its semantic type. Then determination is based on the type of the argument.

20884 **RETURN VALUE**20885 The *isinf()* macro shall return a non-zero value if and only if its argument has an infinite value.20886 **ERRORS**

20887 No errors are defined.

20888 **EXAMPLES**

20889 None.

20890 **APPLICATION USAGE**

20891 None.

20892 **RATIONALE**

20893 None.

20894 **FUTURE DIRECTIONS**

20895 None.

20896 **SEE ALSO**

20897 *fpclassify()*, *isfinite()*, *isnan()*, *isnormal()*, *signbit()*, the Base Definitions volume of
20898 IEEE Std 1003.1-2001 <math.h>

20899 **CHANGE HISTORY**

20900 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

20901 **NAME**

20902 isless — test if x is less than y

20903 **SYNOPSIS**

20904 #include <math.h>

20905 int isless(real-floating x, real-floating y);

20906 **DESCRIPTION**

20907 cx The functionality described on this reference page is aligned with the ISO C standard. Any
20908 conflict between the requirements described here and the ISO C standard is unintentional. This
20909 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

20910 The *isless()* macro shall determine whether its first argument is less than its second argument.
20911 The value of *isless(x, y)* shall be equal to $(x) < (y)$; however, unlike $(x) < (y)$, *isless(x, y)* shall not
20912 raise the invalid floating-point exception when x and y are unordered.

20913 **RETURN VALUE**20914 Upon successful completion, the *isless()* macro shall return the value of $(x) < (y)$.

20915 If x or y is NaN, 0 shall be returned.

20916 **ERRORS**

20917 No errors are defined.

20918 **EXAMPLES**

20919 None.

20920 **APPLICATION USAGE**

20921 The relational and equality operators support the usual mathematical relationships between
20922 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
20923 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
20924 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
20925 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
20926 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
20927 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
20928 indicates that the argument shall be an expression of **real-floating** type.

20929 **RATIONALE**

20930 None.

20931 **FUTURE DIRECTIONS**

20932 None.

20933 **SEE ALSO**

20934 *isgreater()*, *isgreaterequal()*, *islessequal()*, *islessgreater()*, *isunordered()*, the Base Definitions volume
20935 of IEEE Std 1003.1-2001, <math.h>

20936 **CHANGE HISTORY**

20937 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

20938 **NAME**

20939 islessequal — test if x is less than or equal to y

20940 **SYNOPSIS**

20941 #include <math.h>

20942 int islessequal(real-floating x, real-floating y);

20943 **DESCRIPTION**

20944 cx The functionality described on this reference page is aligned with the ISO C standard. Any
20945 conflict between the requirements described here and the ISO C standard is unintentional. This
20946 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

20947 The *islessequal()* macro shall determine whether its first argument is less than or equal to its
20948 second argument. The value of *islessequal*(x, y) shall be equal to (x) <= (y); however, unlike
20949 (x) <= (y), *islessequal*(x, y) shall not raise the invalid floating-point exception when x and y are
20950 unordered.

20951 **RETURN VALUE**20952 Upon successful completion, the *islessequal()* macro shall return the value of (x) <= (y).

20953 If x or y is NaN, 0 shall be returned.

20954 **ERRORS**

20955 No errors are defined.

20956 **EXAMPLES**

20957 None.

20958 **APPLICATION USAGE**

20959 The relational and equality operators support the usual mathematical relationships between
20960 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
20961 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
20962 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
20963 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
20964 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
20965 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
20966 indicates that the argument shall be an expression of **real-floating** type.

20967 **RATIONALE**

20968 None.

20969 **FUTURE DIRECTIONS**

20970 None.

20971 **SEE ALSO**

20972 *isgreater()*, *isgreaterequal()*, *isless()*, *islessgreater()*, *isunordered()*, the Base Definitions volume of
20973 IEEE Std 1003.1-2001 <math.h>

20974 **CHANGE HISTORY**

20975 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

20976 **NAME**20977 *islessgreater* — test if *x* is less than or greater than *y*20978 **SYNOPSIS**20979 `#include <math.h>`20980 `int islessgreater(real-floating x, real-floating y);`20981 **DESCRIPTION**

20982 *CX* The functionality described on this reference page is aligned with the ISO C standard. Any
 20983 conflict between the requirements described here and the ISO C standard is unintentional. This
 20984 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

20985 The *islessgreater()* macro shall determine whether its first argument is less than or greater than
 20986 its second argument. The *islessgreater(x, y)* macro is similar to $(x) < (y) \mid (x) > (y)$; however,
 20987 *islessgreater(x, y)* shall not raise the invalid floating-point exception when *x* and *y* are unordered
 20988 (nor shall it evaluate *x* and *y* twice).

20989 **RETURN VALUE**

20990 Upon successful completion, the *islessgreater()* macro shall return the value of
 20991 $(x) < (y) \mid (x) > (y)$.

20992 If *x* or *y* is NaN, 0 shall be returned.20993 **ERRORS**

20994 No errors are defined.

20995 **EXAMPLES**

20996 None.

20997 **APPLICATION USAGE**

20998 The relational and equality operators support the usual mathematical relationships between
 20999 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
 21000 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
 21001 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
 21002 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
 21003 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
 21004 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
 21005 indicates that the argument shall be an expression of **real-floating** type.

21006 **RATIONALE**

21007 None.

21008 **FUTURE DIRECTIONS**

21009 None.

21010 **SEE ALSO**

21011 *isgreater()*, *isgreaterequal()*, *isless()*, *islessequal()*, *isunordered()*, the Base Definitions volume of
 21012 IEEE Std 1003.1-2001 **<math.h>**

21013 **CHANGE HISTORY**

21014 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

21015 **NAME**

21016 islower — test for a lowercase letter

21017 **SYNOPSIS**

21018 #include <ctype.h>

21019 int islower(int c);

21020 **DESCRIPTION**

21021 cx The functionality described on this reference page is aligned with the ISO C standard. Any
 21022 conflict between the requirements described here and the ISO C standard is unintentional. This
 21023 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21024 The *islower()* function shall test whether *c* is a character of class **lower** in the program's current
 21025 locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

21026 The *c* argument is an **int**, the value of which the application shall ensure is a character
 21027 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
 21028 any other value, the behavior is undefined.

21029 **RETURN VALUE**21030 The *islower()* function shall return non-zero if *c* is a lowercase letter; otherwise, it shall return 0.21031 **ERRORS**

21032 No errors are defined.

21033 **EXAMPLES**21034 **Testing for a Lowercase Letter**

21035 The following example tests whether the value is a lowercase letter, based on the locale of the
 21036 user, then uses it as part of a key value.

```

21037 #include <ctype.h>
21038 #include <stdlib.h>
21039 #include <locale.h>
21040 ...
21041 char *keyst;
21042 int elementlen, len;
21043 char c;
21044 ...
21045 setlocale(LC_ALL, "");
21046 ...
21047 len = 0;
21048 while (len < elementlen) {
21049     c = (char) (rand() % 256);
21050     ...
21051     if (islower(c))
21052         keyst[len++] = c;
21053 }
21054 ...

```

21055 **APPLICATION USAGE**

21056 To ensure applications portability, especially across natural languages, only this function and
 21057 those listed in the SEE ALSO section should be used for character classification.

21058 **RATIONALE**

21059 None.

21060 **FUTURE DIRECTIONS**

21061 None.

21062 **SEE ALSO**

21063 *isalnum()*, *isalpha()*, *iscntrl()*, *isdigit()*, *isgraph()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*,
21064 *isxdigit()*, *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale,
21065 **<ctype.h>**

21066 **CHANGE HISTORY**

21067 First released in Issue 1. Derived from Issue 1 of the SVID.

21068 **Issue 6**

21069 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

21070 An example is added.

21071 **NAME**

21072 isnan — test for a NaN

21073 **SYNOPSIS**

21074 #include <math.h>

21075 int isnan(real-floating x);

21076 **DESCRIPTION**

21077 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
21078 conflict between the requirements described here and the ISO C standard is unintentional. This
21079 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21080 The *isnan()* macro shall determine whether its argument value is a NaN. First, an argument
21081 represented in a format wider than its semantic type is converted to its semantic type. Then
21082 determination is based on the type of the argument.

21083 **RETURN VALUE**21084 The *isnan()* macro shall return a non-zero value if and only if its argument has a NaN value.21085 **ERRORS**

21086 No errors are defined.

21087 **EXAMPLES**

21088 None.

21089 **APPLICATION USAGE**

21090 None.

21091 **RATIONALE**

21092 None.

21093 **FUTURE DIRECTIONS**

21094 None.

21095 **SEE ALSO**

21096 *fpclassify()*, *isfinite()*, *isinf()*, *isnormal()*, *signbit()*, the Base Definitions volume of
21097 IEEE Std 1003.1-2001, <math.h>

21098 **CHANGE HISTORY**

21099 First released in Issue 3.

21100 **Issue 5**

21101 The DESCRIPTION is updated to indicate the return value when NaN is not supported. This
21102 text was previously published in the APPLICATION USAGE section.

21103 **Issue 6**

21104 Entry re-written for alignment with the ISO/IEC 9899:1999 standard.

21105 **NAME**

21106 isnormal — test for a normal value

21107 **SYNOPSIS**

21108 #include <math.h>

21109 int isnormal(real-floating x);

21110 **DESCRIPTION**

21111 cx The functionality described on this reference page is aligned with the ISO C standard. Any
21112 conflict between the requirements described here and the ISO C standard is unintentional. This
21113 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21114 The *isnormal()* macro shall determine whether its argument value is normal (neither zero,
21115 subnormal, infinite, nor NaN). First, an argument represented in a format wider than its
21116 semantic type is converted to its semantic type. Then determination is based on the type of the
21117 argument.

21118 **RETURN VALUE**

21119 The *isnormal()* macro shall return a non-zero value if and only if its argument has a normal
21120 value.

21121 **ERRORS**

21122 No errors are defined.

21123 **EXAMPLES**

21124 None.

21125 **APPLICATION USAGE**

21126 None.

21127 **RATIONALE**

21128 None.

21129 **FUTURE DIRECTIONS**

21130 None.

21131 **SEE ALSO**

21132 *fpclassify()*, *isfinite()*, *isinf()*, *isnan()*, *signbit()*, the Base Definitions volume of
21133 IEEE Std 1003.1-2001, <math.h>

21134 **CHANGE HISTORY**

21135 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

21136 **NAME**

21137 isprint — test for a printable character

21138 **SYNOPSIS**

21139 #include <ctype.h>

21140 int isprint(int c);

21141 **DESCRIPTION**

21142 cx The functionality described on this reference page is aligned with the ISO C standard. Any
21143 conflict between the requirements described here and the ISO C standard is unintentional. This
21144 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21145 The *isprint()* function shall test whether *c* is a character of class **print** in the program's current
21146 locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

21147 The *c* argument is an **int**, the value of which the application shall ensure is a character
21148 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
21149 any other value, the behavior is undefined.

21150 **RETURN VALUE**

21151 The *isprint()* function shall return non-zero if *c* is a printable character; otherwise, it shall return
21152 0.

21153 **ERRORS**

21154 No errors are defined.

21155 **EXAMPLES**

21156 None.

21157 **APPLICATION USAGE**

21158 To ensure applications portability, especially across natural languages, only this function and
21159 those listed in the SEE ALSO section should be used for character classification.

21160 **RATIONALE**

21161 None.

21162 **FUTURE DIRECTIONS**

21163 None.

21164 **SEE ALSO**

21165 *isalnum()*, *isalpha()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *ispunct()*, *isspace()*, *isupper()*,
21166 *isxdigit()*, *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale,
21167 <ctype.h>

21168 **CHANGE HISTORY**

21169 First released in Issue 1. Derived from Issue 1 of the SVID.

21170 **Issue 6**

21171 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

21172 **NAME**

21173 ispunct — test for a punctuation character

21174 **SYNOPSIS**

21175 #include <ctype.h>

21176 int ispunct(int c);

21177 **DESCRIPTION**

21178 cx The functionality described on this reference page is aligned with the ISO C standard. Any
21179 conflict between the requirements described here and the ISO C standard is unintentional. This
21180 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21181 The *ispunct()* function shall test whether *c* is a character of class **punct** in the program's current
21182 locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

21183 The *c* argument is an **int**, the value of which the application shall ensure is a character
21184 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
21185 any other value, the behavior is undefined.

21186 **RETURN VALUE**

21187 The *ispunct()* function shall return non-zero if *c* is a punctuation character; otherwise, it shall
21188 return 0.

21189 **ERRORS**

21190 No errors are defined.

21191 **EXAMPLES**

21192 None.

21193 **APPLICATION USAGE**

21194 To ensure applications portability, especially across natural languages, only this function and
21195 those listed in the SEE ALSO section should be used for character classification.

21196 **RATIONALE**

21197 None.

21198 **FUTURE DIRECTIONS**

21199 None.

21200 **SEE ALSO**

21201 *isalnum()*, *isalpha()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *isspace()*, *isupper()*, *isxdigit()*,
21202 *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <ctype.h>

21203 **CHANGE HISTORY**

21204 First released in Issue 1. Derived from Issue 1 of the SVID.

21205 **Issue 6**

21206 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

21207 **NAME**

21208 isspace — test for a white-space character

21209 **SYNOPSIS**

21210 #include <ctype.h>

21211 int isspace(int c);

21212 **DESCRIPTION**

21213 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
21214 conflict between the requirements described here and the ISO C standard is unintentional. This
21215 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21216 The *isspace()* function shall test whether *c* is a character of class **space** in the program's current
21217 locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

21218 The *c* argument is an **int**, the value of which the application shall ensure is a character
21219 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
21220 any other value, the behavior is undefined.

21221 **RETURN VALUE**

21222 The *isspace()* function shall return non-zero if *c* is a white-space character; otherwise, it shall
21223 return 0.

21224 **ERRORS**

21225 No errors are defined.

21226 **EXAMPLES**

21227 None.

21228 **APPLICATION USAGE**

21229 To ensure applications portability, especially across natural languages, only this function and
21230 those listed in the SEE ALSO section should be used for character classification.

21231 **RATIONALE**

21232 None.

21233 **FUTURE DIRECTIONS**

21234 None.

21235 **SEE ALSO**

21236 *isalnum()*, *isalpha()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isupper()*,
21237 *isxdigit()*, *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale,
21238 <ctype.h>

21239 **CHANGE HISTORY**

21240 First released in Issue 1. Derived from Issue 1 of the SVID.

21241 **Issue 6**

21242 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

21243 **NAME**

21244 isunordered — test if arguments are unordered

21245 **SYNOPSIS**

21246 #include <math.h>

21247 int isunordered(real-floating x, real-floating y);

21248 **DESCRIPTION**

21249 cx The functionality described on this reference page is aligned with the ISO C standard. Any
21250 conflict between the requirements described here and the ISO C standard is unintentional. This
21251 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21252 The *isunordered()* macro shall determine whether its arguments are unordered.21253 **RETURN VALUE**

21254 Upon successful completion, the *isunordered()* macro shall return 1 if its arguments are
21255 unordered, and 0 otherwise.

21256 If x or y is NaN, 0 shall be returned.

21257 **ERRORS**

21258 No errors are defined.

21259 **EXAMPLES**

21260 None.

21261 **APPLICATION USAGE**

21262 The relational and equality operators support the usual mathematical relationships between
21263 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
21264 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
21265 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
21266 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
21267 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
21268 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
21269 indicates that the argument shall be an expression of **real-floating** type.

21270 **RATIONALE**

21271 None.

21272 **FUTURE DIRECTIONS**

21273 None.

21274 **SEE ALSO**

21275 *isgreater()*, *isgreaterequal()*, *isless()*, *islessequal()*, *islessgreater()*, the Base Definitions volume of
21276 IEEE Std 1003.1-2001, <math.h>

21277 **CHANGE HISTORY**

21278 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

21279 **NAME**

21280 isupper — test for an uppercase letter

21281 **SYNOPSIS**

21282 #include <ctype.h>

21283 int isupper(int c);

21284 **DESCRIPTION**

21285 cx The functionality described on this reference page is aligned with the ISO C standard. Any
21286 conflict between the requirements described here and the ISO C standard is unintentional. This
21287 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21288 The *isupper()* function shall test whether *c* is a character of class **upper** in the program's current
21289 locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

21290 The *c* argument is an **int**, the value of which the application shall ensure is a character
21291 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
21292 any other value, the behavior is undefined.

21293 **RETURN VALUE**21294 The *isupper()* function shall return non-zero if *c* is an uppercase letter; otherwise, it shall return 0.21295 **ERRORS**

21296 No errors are defined.

21297 **EXAMPLES**

21298 None.

21299 **APPLICATION USAGE**

21300 To ensure applications portability, especially across natural languages, only this function and
21301 those listed in the SEE ALSO section should be used for character classification.

21302 **RATIONALE**

21303 None.

21304 **FUTURE DIRECTIONS**

21305 None.

21306 **SEE ALSO**

21307 *isalnum()*, *isalpha()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isxdigit()*,
21308 *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <ctype.h>

21309 **CHANGE HISTORY**

21310 First released in Issue 1. Derived from Issue 1 of the SVID.

21311 **Issue 6**

21312 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

21313 **NAME**

21314 iswalnum — test for an alphanumeric wide-character code

21315 **SYNOPSIS**

21316 #include <wctype.h>

21317 int iswalnum(wint_t wc);

21318 **DESCRIPTION**

21319 cx The functionality described on this reference page is aligned with the ISO C standard. Any
 21320 conflict between the requirements described here and the ISO C standard is unintentional. This
 21321 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21322 The *iswalnum()* function shall test whether *wc* is a wide-character code representing a character
 21323 of class **alpha** or **digit** in the program's current locale; see the Base Definitions volume of
 21324 IEEE Std 1003.1-2001, Chapter 7, Locale.

21325 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
 21326 code corresponding to a valid character in the current locale, or equal to the value of the macro
 21327 WEOF. If the argument has any other value, the behavior is undefined.

21328 **RETURN VALUE**

21329 The *iswalnum()* function shall return non-zero if *wc* is an alphanumeric wide-character code;
 21330 otherwise, it shall return 0.

21331 **ERRORS**

21332 No errors are defined.

21333 **EXAMPLES**

21334 None.

21335 **APPLICATION USAGE**

21336 To ensure applications portability, especially across natural languages, only this function and
 21337 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21338 **RATIONALE**

21339 None.

21340 **FUTURE DIRECTIONS**

21341 None.

21342 **SEE ALSO**

21343 *iswalphabet()*, *iswcntrl()*, *iswctype()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*, *iswpunct()*,
 21344 *iswspace()*, *iswupper()*, *iswxdigit()*, *setlocale()*, the Base Definitions volume of
 21345 IEEE Std 1003.1-2001, Chapter 7, Locale, <stdio.h>, <wchar.h>, <wctype.h>

21346 **CHANGE HISTORY**

21347 First released as a World-wide Portability Interface in Issue 4.

21348 **Issue 5**

21349 The following change has been made in this issue for alignment with
 21350 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 21351 • The SYNOPSIS has been changed to indicate that this function and associated data types are
 21352 now made visible by inclusion of the <wctype.h> header rather than <wchar.h>.

21353 **Issue 6**

21354 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

21355 **NAME**

21356 iswalpha — test for an alphabetic wide-character code

21357 **SYNOPSIS**

21358 #include <wctype.h>

21359 int iswalpha(wint_t wc);

21360 **DESCRIPTION**

21361 cx The functionality described on this reference page is aligned with the ISO C standard. Any
21362 conflict between the requirements described here and the ISO C standard is unintentional. This
21363 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21364 The *iswalpha()* function shall test whether *wc* is a wide-character code representing a character of
21365 class **alpha** in the program's current locale; see the Base Definitions volume of
21366 IEEE Std 1003.1-2001, Chapter 7, Locale.

21367 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
21368 code corresponding to a valid character in the current locale, or equal to the value of the macro
21369 WEOF. If the argument has any other value, the behavior is undefined.

21370 **RETURN VALUE**

21371 The *iswalpha()* function shall return non-zero if *wc* is an alphabetic wide-character code;
21372 otherwise, it shall return 0.

21373 **ERRORS**

21374 No errors are defined.

21375 **EXAMPLES**

21376 None.

21377 **APPLICATION USAGE**

21378 To ensure applications portability, especially across natural languages, only this function and
21379 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21380 **RATIONALE**

21381 None.

21382 **FUTURE DIRECTIONS**

21383 None.

21384 **SEE ALSO**

21385 *iswalnum()*, *iswcntrl()*, *iswctype()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*, *iswpunct()*,
21386 *iswspace()*, *iswupper()*, *iswxdigit()*, *setlocale()*, the Base Definitions volume of
21387 IEEE Std 1003.1-2001, Chapter 7, Locale, <stdio.h>, <wchar.h>, <wctype.h>

21388 **CHANGE HISTORY**

21389 First released in Issue 4.

21390 **Issue 5**

21391 The following change has been made in this issue for alignment with
21392 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 21393 • The SYNOPSIS has been changed to indicate that this function and associated data types are
21394 now made visible by inclusion of the <wctype.h> header rather than <wchar.h>.

21395 **Issue 6**

21396 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

21397 **NAME**

21398 iswblank — test for a blank wide-character code

21399 **SYNOPSIS**

21400 #include <wctype.h>

21401 int iswblank(wint_t wc);

21402 **DESCRIPTION**

21403 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
21404 conflict between the requirements described here and the ISO C standard is unintentional. This
21405 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21406 The *iswblank()* function shall test whether *wc* is a wide-character code representing a character of
21407 class **blank** in the program's current locale; see the Base Definitions volume of
21408 IEEE Std 1003.1-2001, Chapter 7, Locale.

21409 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
21410 code corresponding to a valid character in the current locale, or equal to the value of the macro
21411 WEOF. If the argument has any other value, the behavior is undefined.

21412 **RETURN VALUE**

21413 The *iswblank()* function shall return non-zero if *wc* is a blank wide-character code; otherwise, it
21414 shall return 0.

21415 **ERRORS**

21416 No errors are defined.

21417 **EXAMPLES**

21418 None.

21419 **APPLICATION USAGE**

21420 To ensure applications portability, especially across natural languages, only this function and
21421 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21422 **RATIONALE**

21423 None.

21424 **FUTURE DIRECTIONS**

21425 None.

21426 **SEE ALSO**

21427 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswctype()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*,
21428 *iswpunct()*, *iswspace()*, *iswupper()*, *iswxdigit()*, *setlocale()*, the Base Definitions volume of
21429 IEEE Std 1003.1-2001, Chapter 7, Locale, <stdio.h>, <wchar.h>, <wctype.h>

21430 **CHANGE HISTORY**

21431 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

21432 **NAME**

21433 iswcntrl — test for a control wide-character code

21434 **SYNOPSIS**

21435 #include <wctype.h>

21436 int iswcntrl(wint_t wc);

21437 **DESCRIPTION**

21438 cx The functionality described on this reference page is aligned with the ISO C standard. Any
21439 conflict between the requirements described here and the ISO C standard is unintentional. This
21440 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21441 The *iswcntrl()* function shall test whether *wc* is a wide-character code representing a character of
21442 class **cntrl** in the program's current locale; see the Base Definitions volume of
21443 IEEE Std 1003.1-2001, Chapter 7, Locale.

21444 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
21445 code corresponding to a valid character in the current locale, or equal to the value of the macro
21446 WEOF. If the argument has any other value, the behavior is undefined.

21447 **RETURN VALUE**

21448 The *iswcntrl()* function shall return non-zero if *wc* is a control wide-character code; otherwise, it
21449 shall return 0.

21450 **ERRORS**

21451 No errors are defined.

21452 **EXAMPLES**

21453 None.

21454 **APPLICATION USAGE**

21455 To ensure applications portability, especially across natural languages, only this function and
21456 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21457 **RATIONALE**

21458 None.

21459 **FUTURE DIRECTIONS**

21460 None.

21461 **SEE ALSO**

21462 *iswalnum()*, *iswalpha()*, *iswctype()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*, *iswpunct()*,
21463 *iswspace()*, *iswupper()*, *iswxdigit()*, *setlocale()*, the Base Definitions volume of
21464 IEEE Std 1003.1-2001, Chapter 7, Locale, <**wchar.h**>, <**wctype.h**>

21465 **CHANGE HISTORY**

21466 First released in Issue 4.

21467 **Issue 5**

21468 The following change has been made in this issue for alignment with
21469 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 21470 • The SYNOPSIS has been changed to indicate that this function and associated data types are
21471 now made visible by inclusion of the <**wctype.h**> header rather than <**wchar.h**>.

21472 **Issue 6**

21473 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

21474 **NAME**

21475 iswctype — test character for a specified class

21476 **SYNOPSIS**

21477 #include <wctype.h>

21478 int iswctype(wint_t *wc*, wctype_t *charclass*);21479 **DESCRIPTION**

21480 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 21481 conflict between the requirements described here and the ISO C standard is unintentional. This
 21482 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21483 The *iswctype()* function shall determine whether the wide-character code *wc* has the character
 21484 class *charclass*, returning true or false. The *iswctype()* function is defined on WEOF and wide-
 21485 character codes corresponding to the valid character encodings in the current locale. If the *wc*
 21486 argument is not in the domain of the function, the result is undefined. If the value of *charclass* is
 21487 invalid (that is, not obtained by a call to *wctype()* or *charclass* is invalidated by a subsequent call
 21488 to *setlocale()* that has affected category *LC_CTYPE*) the result is unspecified.

21489 **RETURN VALUE**

21490 The *iswctype()* function shall return non-zero (true) if and only if *wc* has the property described
 21491 CX by *charclass*. If *charclass* is 0, *iswctype()* shall return 0.

21492 **ERRORS**

21493 No errors are defined.

21494 **EXAMPLES**21495 **Testing for a Valid Character**

```
21496 #include <wctype.h>
21497 ...
21498 int yes_or_no;
21499 wint_t wc;
21500 wctype_t valid_class;
21501 ...
21502 if ((valid_class=wctype("vowel")) == (wctype_t)0)
21503     /* Invalid character class. */
21504     yes_or_no=iswctype(wc,valid_class);
```

21505 **APPLICATION USAGE**

21506 The twelve strings "alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower",
 21507 "print", "punct", "space", "upper", and "xdigit" are reserved for the standard
 21508 character classes. In the table below, the functions in the left column are equivalent to the
 21509 functions in the right column.

21510 iswalnum(<i>wc</i>)	iswctype(<i>wc</i> , wctype("alnum"))
21511 iswalpha(<i>wc</i>)	iswctype(<i>wc</i> , wctype("alpha"))
21512 iswblank(<i>wc</i>)	iswctype(<i>wc</i> , wctype("blank"))
21513 iswcntrl(<i>wc</i>)	iswctype(<i>wc</i> , wctype("cntrl"))
21514 iswdigit(<i>wc</i>)	iswctype(<i>wc</i> , wctype("digit"))
21515 iswgraph(<i>wc</i>)	iswctype(<i>wc</i> , wctype("graph"))
21516 iswlower(<i>wc</i>)	iswctype(<i>wc</i> , wctype("lower"))
21517 iswprint(<i>wc</i>)	iswctype(<i>wc</i> , wctype("print"))
21518 iswpunct(<i>wc</i>)	iswctype(<i>wc</i> , wctype("punct"))
21519 iswspace(<i>wc</i>)	iswctype(<i>wc</i> , wctype("space"))

21520 `iswupper(wc)` `iswctype(wc, wctype("upper"))`
21521 `iswxdigit(wc)` `iswctype(wc, wctype("xdigit"))`

21522 **RATIONALE**

21523 None.

21524 **FUTURE DIRECTIONS**

21525 None.

21526 **SEE ALSO**

21527 `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`,
21528 `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `wctype()`, the Base Definitions volume of
21529 IEEE Std 1003.1-2001, `<wchar.h>`, `<wctype.h>`

21530 **CHANGE HISTORY**

21531 First released as World-wide Portability Interfaces in Issue 4.

21532 **Issue 5**

21533 The following change has been made in this issue for alignment with
21534 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 21535 • The SYNOPSIS has been changed to indicate that this function and associated data types are
21536 now made visible by inclusion of the `<wctype.h>` header rather than `<wchar.h>`.

21537 **Issue 6**

21538 The behavior of $n=0$ is now described.

21539 An example is added.

21540 A new function, `iswblank()`, is added to the list in the APPLICATION USAGE.

21541 **NAME**

21542 iswdigit — test for a decimal digit wide-character code

21543 **SYNOPSIS**

21544 #include <wctype.h>

21545 int iswdigit(wint_t wc);

21546 **DESCRIPTION**

21547 cx The functionality described on this reference page is aligned with the ISO C standard. Any
 21548 conflict between the requirements described here and the ISO C standard is unintentional. This
 21549 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21550 The *iswdigit()* function shall test whether *wc* is a wide-character code representing a character of
 21551 class **digit** in the program's current locale; see the Base Definitions volume of
 21552 IEEE Std 1003.1-2001, Chapter 7, Locale.

21553 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
 21554 code corresponding to a valid character in the current locale, or equal to the value of the macro
 21555 WEOF. If the argument has any other value, the behavior is undefined.

21556 **RETURN VALUE**

21557 The *iswdigit()* function shall return non-zero if *wc* is a decimal digit wide-character code;
 21558 otherwise, it shall return 0.

21559 **ERRORS**

21560 No errors are defined.

21561 **EXAMPLES**

21562 None.

21563 **APPLICATION USAGE**

21564 To ensure applications portability, especially across natural languages, only this function and
 21565 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21566 **RATIONALE**

21567 None.

21568 **FUTURE DIRECTIONS**

21569 None.

21570 **SEE ALSO**

21571 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswctype()*, *iswgraph()*, *iswlower()*, *iswprint()*, *iswpunct()*,
 21572 *iswspace()*, *iswupper()*, *iswxdigit()*, *setlocale()*, the Base Definitions volume of
 21573 IEEE Std 1003.1-2001, Chapter 7, Locale, <wchar.h>, <wctype.h>

21574 **CHANGE HISTORY**

21575 First released in Issue 4.

21576 **Issue 5**

21577 The following change has been made in this issue for alignment with
 21578 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 21579 • The SYNOPSIS has been changed to indicate that this function and associated data types are
 21580 now made visible by inclusion of the <wctype.h> header rather than <wchar.h>.

21581 **Issue 6**

21582 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

21583 **NAME**

21584 iswgraph — test for a visible wide-character code

21585 **SYNOPSIS**

21586 #include <wctype.h>

21587 int iswgraph(wint_t wc);

21588 **DESCRIPTION**

21589 cx The functionality described on this reference page is aligned with the ISO C standard. Any
21590 conflict between the requirements described here and the ISO C standard is unintentional. This
21591 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21592 The *iswgraph()* function shall test whether *wc* is a wide-character code representing a character
21593 of class **graph** in the program's current locale; see the Base Definitions volume of
21594 IEEE Std 1003.1-2001, Chapter 7, Locale.

21595 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
21596 code corresponding to a valid character in the current locale, or equal to the value of the macro
21597 WEOF. If the argument has any other value, the behavior is undefined.

21598 **RETURN VALUE**

21599 The *iswgraph()* function shall return non-zero if *wc* is a wide-character code with a visible
21600 representation; otherwise, it shall return 0.

21601 **ERRORS**

21602 No errors are defined.

21603 **EXAMPLES**

21604 None.

21605 **APPLICATION USAGE**

21606 To ensure applications portability, especially across natural languages, only this function and
21607 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21608 **RATIONALE**

21609 None.

21610 **FUTURE DIRECTIONS**

21611 None.

21612 **SEE ALSO**

21613 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswctype()*, *iswdigit()*, *iswlower()*, *iswprint()*, *iswpunct()*,
21614 *iswspace()*, *iswupper()*, *iswxdigit()*, *setlocale()*, the Base Definitions volume of
21615 IEEE Std 1003.1-2001, Chapter 7, Locale, <wchar.h>, <wctype.h>

21616 **CHANGE HISTORY**

21617 First released in Issue 4.

21618 **Issue 5**

21619 The following change has been made in this issue for alignment with
21620 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 21621 • The SYNOPSIS has been changed to indicate that this function and associated data types are
21622 now made visible by inclusion of the <wctype.h> header rather than <wchar.h>.

21623 **Issue 6**

21624 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

21625 **NAME**

21626 iswlower — test for a lowercase letter wide-character code

21627 **SYNOPSIS**

21628 #include <wctype.h>

21629 int iswlower(wint_t wc);

21630 **DESCRIPTION**

21631 cx The functionality described on this reference page is aligned with the ISO C standard. Any
 21632 conflict between the requirements described here and the ISO C standard is unintentional. This
 21633 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21634 The *iswlower()* function shall test whether *wc* is a wide-character code representing a character
 21635 of class **lower** in the program's current locale; see the Base Definitions volume of
 21636 IEEE Std 1003.1-2001, Chapter 7, Locale.

21637 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
 21638 code corresponding to a valid character in the current locale, or equal to the value of the macro
 21639 WEOF. If the argument has any other value, the behavior is undefined.

21640 **RETURN VALUE**

21641 The *iswlower()* function shall return non-zero if *wc* is a lowercase letter wide-character code;
 21642 otherwise, it shall return 0.

21643 **ERRORS**

21644 No errors are defined.

21645 **EXAMPLES**

21646 None.

21647 **APPLICATION USAGE**

21648 To ensure applications portability, especially across natural languages, only this function and
 21649 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21650 **RATIONALE**

21651 None.

21652 **FUTURE DIRECTIONS**

21653 None.

21654 **SEE ALSO**

21655 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswctype()*, *iswdigit()*, *iswgraph()*, *iswprint()*, *iswpunct()*,
 21656 *iswspace()*, *iswupper()*, *iswxdigit()*, *setlocale()*, the Base Definitions volume of
 21657 IEEE Std 1003.1-2001, Chapter 7, Locale, <wchar.h>, <wctype.h>

21658 **CHANGE HISTORY**

21659 First released in Issue 4.

21660 **Issue 5**

21661 The following change has been made in this issue for alignment with
 21662 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 21663 • The SYNOPSIS has been changed to indicate that this function and associated data types are
 21664 now made visible by inclusion of the <wctype.h> header rather than <wchar.h>.

21665 **Issue 6**

21666 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

21667 **NAME**

21668 iswprint — test for a printable wide-character code

21669 **SYNOPSIS**

21670 #include <wctype.h>

21671 int iswprint(wint_t wc);

21672 **DESCRIPTION**

21673 cx The functionality described on this reference page is aligned with the ISO C standard. Any
21674 conflict between the requirements described here and the ISO C standard is unintentional. This
21675 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21676 The *iswprint()* function shall test whether *wc* is a wide-character code representing a character of
21677 class **print** in the program's current locale; see the Base Definitions volume of
21678 IEEE Std 1003.1-2001, Chapter 7, Locale.

21679 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
21680 code corresponding to a valid character in the current locale, or equal to the value of the macro
21681 WEOF. If the argument has any other value, the behavior is undefined.

21682 **RETURN VALUE**

21683 The *iswprint()* function shall return non-zero if *wc* is a printable wide-character code; otherwise,
21684 it shall return 0.

21685 **ERRORS**

21686 No errors are defined.

21687 **EXAMPLES**

21688 None.

21689 **APPLICATION USAGE**

21690 To ensure applications portability, especially across natural languages, only this function and
21691 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21692 **RATIONALE**

21693 None.

21694 **FUTURE DIRECTIONS**

21695 None.

21696 **SEE ALSO**

21697 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswctype()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswpunct()*,
21698 *iswspace()*, *iswupper()*, *iswxdigit()*, *setlocale()*, the Base Definitions volume of
21699 IEEE Std 1003.1-2001, Chapter 7, Locale, <wchar.h>, <wctype.h>

21700 **CHANGE HISTORY**

21701 First released in Issue 4.

21702 **Issue 5**

21703 The following change has been made in this issue for alignment with
21704 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 21705 • The SYNOPSIS has been changed to indicate that this function and associated data types are
21706 now made visible by inclusion of the <wctype.h> header rather than <wchar.h>.

21707 **Issue 6**

21708 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

21709 **NAME**

21710 iswpunct — test for a punctuation wide-character code

21711 **SYNOPSIS**

21712 #include <wctype.h>

21713 int iswpunct(wint_t wc);

21714 **DESCRIPTION**

21715 cx The functionality described on this reference page is aligned with the ISO C standard. Any
 21716 conflict between the requirements described here and the ISO C standard is unintentional. This
 21717 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21718 The *iswpunct()* function shall test whether *wc* is a wide-character code representing a character
 21719 of class **punct** in the program's current locale; see the Base Definitions volume of
 21720 IEEE Std 1003.1-2001, Chapter 7, Locale.

21721 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
 21722 code corresponding to a valid character in the current locale, or equal to the value of the macro
 21723 WEOF. If the argument has any other value, the behavior is undefined.

21724 **RETURN VALUE**

21725 The *iswpunct()* function shall return non-zero if *wc* is a punctuation wide-character code;
 21726 otherwise, it shall return 0.

21727 **ERRORS**

21728 No errors are defined.

21729 **EXAMPLES**

21730 None.

21731 **APPLICATION USAGE**

21732 To ensure applications portability, especially across natural languages, only this function and
 21733 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21734 **RATIONALE**

21735 None.

21736 **FUTURE DIRECTIONS**

21737 None.

21738 **SEE ALSO**

21739 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswctype()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*,
 21740 *iswspace()*, *iswupper()*, *iswxdigit()*, *setlocale()*, the Base Definitions volume of
 21741 IEEE Std 1003.1-2001, Chapter 7, Locale, <**wchar.h**>, <**wctype.h**>

21742 **CHANGE HISTORY**

21743 First released in Issue 4.

21744 **Issue 5**

21745 The following change has been made in this issue for alignment with
 21746 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 21747 • The SYNOPSIS has been changed to indicate that this function and associated data types are
 21748 now made visible by inclusion of the <**wctype.h**> header rather than <**wchar.h**>.

21749 **Issue 6**

21750 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

21751 **NAME**

21752 iswspace — test for a white-space wide-character code

21753 **SYNOPSIS**

21754 #include <wctype.h>

21755 int iswspace(wint_t wc);

21756 **DESCRIPTION**

21757 cx The functionality described on this reference page is aligned with the ISO C standard. Any
21758 conflict between the requirements described here and the ISO C standard is unintentional. This
21759 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21760 The *iswspace()* function shall test whether *wc* is a wide-character code representing a character of
21761 class **space** in the program's current locale; see the Base Definitions volume of
21762 IEEE Std 1003.1-2001, Chapter 7, Locale.

21763 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
21764 code corresponding to a valid character in the current locale, or equal to the value of the macro
21765 WEOF. If the argument has any other value, the behavior is undefined.

21766 **RETURN VALUE**

21767 The *iswspace()* function shall return non-zero if *wc* is a white-space wide-character code;
21768 otherwise, it shall return 0.

21769 **ERRORS**

21770 No errors are defined.

21771 **EXAMPLES**

21772 None.

21773 **APPLICATION USAGE**

21774 To ensure applications portability, especially across natural languages, only this function and
21775 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21776 **RATIONALE**

21777 None.

21778 **FUTURE DIRECTIONS**

21779 None.

21780 **SEE ALSO**

21781 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswctype()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*,
21782 *iswpunct()*, *iswupper()*, *iswxdigit()*, *setlocale()*, the Base Definitions volume of
21783 IEEE Std 1003.1-2001, Chapter 7, Locale, <**wchar.h**>, <**wctype.h**>

21784 **CHANGE HISTORY**

21785 First released in Issue 4.

21786 **Issue 5**

21787 The following change has been made in this issue for alignment with
21788 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 21789 • The SYNOPSIS has been changed to indicate that this function and associated data types are
21790 now made visible by inclusion of the <**wctype.h**> header rather than <**wchar.h**>.

21791 **Issue 6**

21792 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

21793 **NAME**

21794 iswupper — test for an uppercase letter wide-character code

21795 **SYNOPSIS**

21796 #include <wctype.h>

21797 int iswupper(wint_t wc);

21798 **DESCRIPTION**

21799 cx The functionality described on this reference page is aligned with the ISO C standard. Any
 21800 conflict between the requirements described here and the ISO C standard is unintentional. This
 21801 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21802 The *iswupper()* function shall test whether *wc* is a wide-character code representing a character
 21803 of class **upper** in the program's current locale; see the Base Definitions volume of
 21804 IEEE Std 1003.1-2001, Chapter 7, Locale.

21805 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
 21806 code corresponding to a valid character in the current locale, or equal to the value of the macro
 21807 WEOF. If the argument has any other value, the behavior is undefined.

21808 **RETURN VALUE**

21809 The *iswupper()* function shall return non-zero if *wc* is an uppercase letter wide-character code;
 21810 otherwise, it shall return 0.

21811 **ERRORS**

21812 No errors are defined.

21813 **EXAMPLES**

21814 None.

21815 **APPLICATION USAGE**

21816 To ensure applications portability, especially across natural languages, only this function and
 21817 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21818 **RATIONALE**

21819 None.

21820 **FUTURE DIRECTIONS**

21821 None.

21822 **SEE ALSO**

21823 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswctype()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*,
 21824 *iswpunct()*, *iswspace()*, *iswxdigit()*, *setlocale()*, the Base Definitions volume of
 21825 IEEE Std 1003.1-2001, Chapter 7, Locale, <wchar.h>, <wctype.h>

21826 **CHANGE HISTORY**

21827 First released in Issue 4.

21828 **Issue 5**

21829 The following change has been made in this issue for alignment with
 21830 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 21831 • The SYNOPSIS has been changed to indicate that this function and associated data types are
 21832 now made visible by inclusion of the <wctype.h> header rather than <wchar.h>.

21833 **Issue 6**

21834 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

21835 **NAME**

21836 iswxdigit — test for a hexadecimal digit wide-character code

21837 **SYNOPSIS**

21838 #include <wctype.h>

21839 int iswxdigit(wint_t wc);

21840 **DESCRIPTION**

21841 cx The functionality described on this reference page is aligned with the ISO C standard. Any
21842 conflict between the requirements described here and the ISO C standard is unintentional. This
21843 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21844 The *iswxdigit()* function shall test whether *wc* is a wide-character code representing a character
21845 of class **xdigit** in the program's current locale; see the Base Definitions volume of
21846 IEEE Std 1003.1-2001, Chapter 7, Locale.

21847 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
21848 code corresponding to a valid character in the current locale, or equal to the value of the macro
21849 WEOF. If the argument has any other value, the behavior is undefined.

21850 **RETURN VALUE**

21851 The *iswxdigit()* function shall return non-zero if *wc* is a hexadecimal digit wide-character code;
21852 otherwise, it shall return 0.

21853 **ERRORS**

21854 No errors are defined.

21855 **EXAMPLES**

21856 None.

21857 **APPLICATION USAGE**

21858 To ensure applications portability, especially across natural languages, only this function and
21859 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21860 **RATIONALE**

21861 None.

21862 **FUTURE DIRECTIONS**

21863 None.

21864 **SEE ALSO**

21865 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswctype()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*,
21866 *iswpunct()*, *iswspace()*, *iswupper()*, *setlocale()*, the Base Definitions volume of
21867 IEEE Std 1003.1-2001, Chapter 7, Locale, <**wchar.h**>, <**wctype.h**>

21868 **CHANGE HISTORY**

21869 First released in Issue 4.

21870 **Issue 5**

21871 The following change has been made in this issue for alignment with
21872 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 21873 • The SYNOPSIS has been changed to indicate that this function and associated data types are
21874 now made visible by inclusion of the <**wctype.h**> header rather than <**wchar.h**>.

21875 **Issue 6**

21876 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

21877 **NAME**

21878 isxdigit — test for a hexadecimal digit

21879 **SYNOPSIS**

21880 #include <ctype.h>

21881 int isxdigit(int c);

21882 **DESCRIPTION**

21883 cx The functionality described on this reference page is aligned with the ISO C standard. Any
21884 conflict between the requirements described here and the ISO C standard is unintentional. This
21885 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21886 The *isxdigit()* function shall test whether *c* is a character of class **xdigit** in the program's current
21887 locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

21888 The *c* argument is an **int**, the value of which the application shall ensure is a character
21889 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
21890 any other value, the behavior is undefined.

21891 **RETURN VALUE**

21892 The *isxdigit()* function shall return non-zero if *c* is a hexadecimal digit; otherwise, it shall return
21893 0.

21894 **ERRORS**

21895 No errors are defined.

21896 **EXAMPLES**

21897 None.

21898 **APPLICATION USAGE**

21899 To ensure applications portability, especially across natural languages, only this function and
21900 those listed in the SEE ALSO section should be used for character classification.

21901 **RATIONALE**

21902 None.

21903 **FUTURE DIRECTIONS**

21904 None.

21905 **SEE ALSO**

21906 *isalnum()*, *isalpha()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*,
21907 the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <ctype.h>

21908 **CHANGE HISTORY**

21909 First released in Issue 1. Derived from Issue 1 of the SVID.

21910 **Issue 6**

21911 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

21912 **NAME**

21913 j0, j1, jn — Bessel functions of the first kind

21914 **SYNOPSIS**

```
21915 xSI      #include <math.h>

21916          double j0(double x);
21917          double j1(double x);
21918          double jn(int n, double x);
21919
```

21920 **DESCRIPTION**

21921 The *j0()*, *j1()*, and *jn()* functions shall compute Bessel functions of *x* of the first kind of orders 0,
 21922 1, and *n*, respectively.

21923 An application wishing to check for error situations should set *errno* to zero and call
 21924 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 21925 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 21926 zero, an error has occurred.

21927 **RETURN VALUE**

21928 Upon successful completion, these functions shall return the relevant Bessel value of *x* of the
 21929 first kind.

21930 If the *x* argument is too large in magnitude, or the correct result would cause underflow, 0 shall
 21931 be returned and a range error may occur.

21932 If *x* is NaN, a NaN shall be returned.

21933 **ERRORS**

21934 These functions may fail if:

21935	Range Error	The value of <i>x</i> was too large in magnitude, or an underflow occurred.
21936		If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
21937		then <i>errno</i> shall be set to [ERANGE]. If the integer expression
21938		(math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
21939		floating-point exception shall be raised.

21940 No other errors shall occur.

21941 **EXAMPLES**

21942 None.

21943 **APPLICATION USAGE**

21944 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
 21945 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

21946 **RATIONALE**

21947 None.

21948 **FUTURE DIRECTIONS**

21949 None.

21950 **SEE ALSO**

21951 *feclearexcept()*, *fetestexcept()*, *isnan()*, *y0()*, the Base Definitions volume of IEEE Std 1003.1-2001,
 21952 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

21953 **CHANGE HISTORY**

21954 First released in Issue 1. Derived from Issue 1 of the SVID.

21955 **Issue 5**

21956 The DESCRIPTION is updated to indicate how an application should check for an error. This
21957 text was previously published in the APPLICATION USAGE section.

21958 **Issue 6**

21959 The may fail [EDOM] error is removed for the case for NaN.

21960 The RETURN VALUE and ERRORS sections are reworked for alignment of the error handling
21961 with the ISO/IEC 9899:1999 standard.

21962 NAME

21963 jrand48 — generate a uniformly distributed pseudo-random long signed integer

21964 SYNOPSIS

21965 xSI #include <stdlib.h>

21966 long jrand48(unsigned short xsubi[3]);

21967

21968 DESCRIPTION

21969 Refer to *drand48()*.

21970 **NAME**

21971 kill — send a signal to a process or a group of processes

21972 **SYNOPSIS**21973 **CX** #include <signal.h>

21974 int kill(pid_t pid, int sig);

21975

21976 **DESCRIPTION**

21977 The *kill()* function shall send a signal to a process or a group of processes specified by *pid*. The
 21978 signal to be sent is specified by *sig* and is either one from the list given in <signal.h> or 0. If *sig* is
 21979 0 (the null signal), error checking is performed but no signal is actually sent. The null signal can
 21980 be used to check the validity of *pid*.

21981 For a process to have permission to send a signal to a process designated by *pid*, unless the
 21982 sending process has appropriate privileges, the real or effective user ID of the sending process
 21983 shall match the real or saved set-user-ID of the receiving process.

21984 If *pid* is greater than 0, *sig* shall be sent to the process whose process ID is equal to *pid*.

21985 If *pid* is 0, *sig* shall be sent to all processes (excluding an unspecified set of system processes)
 21986 whose process group ID is equal to the process group ID of the sender, and for which the
 21987 process has permission to send a signal.

21988 If *pid* is -1, *sig* shall be sent to all processes (excluding an unspecified set of system processes) for
 21989 which the process has permission to send that signal.

21990 If *pid* is negative, but not -1, *sig* shall be sent to all processes (excluding an unspecified set of
 21991 system processes) whose process group ID is equal to the absolute value of *pid*, and for which
 21992 the process has permission to send a signal.

21993 If the value of *pid* causes *sig* to be generated for the sending process, and if *sig* is not blocked for
 21994 the calling thread and if no other thread has *sig* unblocked or is waiting in a *sigwait()* function
 21995 for *sig*, either *sig* or at least one pending unblocked signal shall be delivered to the sending
 21996 thread before *kill()* returns.

21997 The user ID tests described above shall not be applied when sending SIGCONT to a process that
 21998 is a member of the same session as the sending process.

21999 An implementation that provides extended security controls may impose further
 22000 implementation-defined restrictions on the sending of signals, including the null signal. In
 22001 particular, the system may deny the existence of some or all of the processes specified by *pid*.

22002 The *kill()* function is successful if the process has permission to send *sig* to any of the processes
 22003 specified by *pid*. If *kill()* fails, no signal shall be sent.

22004 **RETURN VALUE**

22005 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
 22006 indicate the error.

22007 **ERRORS**22008 The *kill()* function shall fail if:

22009 [EINVAL] The value of the *sig* argument is an invalid or unsupported signal number.

22010 [EPERM] The process does not have permission to send the signal to any receiving
 22011 process.

22012 [ESRCH] No process or process group can be found corresponding to that specified by
 22013 *pid*.

22014 **EXAMPLES**

22015 None.

22016 **APPLICATION USAGE**

22017 None.

22018 **RATIONALE**

22019 The semantics for permission checking for *kill()* differed between System V and most other
22020 implementations, such as Version 7 or 4.3 BSD. The semantics chosen for this volume of
22021 IEEE Std 1003.1-2001 agree with System V. Specifically, a set-user-ID process cannot protect
22022 itself against signals (or at least not against SIGKILL) unless it changes its real user ID. This
22023 choice allows the user who starts an application to send it signals even if it changes its effective
22024 user ID. The other semantics give more power to an application that wants to protect itself from
22025 the user who ran it.

22026 Some implementations provide semantic extensions to the *kill()* function when the absolute
22027 value of *pid* is greater than some maximum, or otherwise special, value. Negative values are a
22028 flag to *kill()*. Since most implementations return [ESRCH] in this case, this behavior is not
22029 included in this volume of IEEE Std 1003.1-2001, although a conforming implementation could
22030 provide such an extension.

22031 The implementation-defined processes to which a signal cannot be sent may include the
22032 scheduler or *init*.

22033 There was initially strong sentiment to specify that, if *pid* specifies that a signal be sent to the
22034 calling process and that signal is not blocked, that signal would be delivered before *kill()*
22035 returns. This would permit a process to call *kill()* and be guaranteed that the call never return.
22036 However, historical implementations that provide only the *signal()* function make only the
22037 weaker guarantee in this volume of IEEE Std 1003.1-2001, because they only deliver one signal
22038 each time a process enters the kernel. Modifications to such implementations to support the
22039 *sigaction()* function generally require entry to the kernel following return from a signal-catching
22040 function, in order to restore the signal mask. Such modifications have the effect of satisfying the
22041 stronger requirement, at least when *sigaction()* is used, but not necessarily when *signal()* is used.
22042 The developers of this volume of IEEE Std 1003.1-2001 considered making the stronger
22043 requirement except when *signal()* is used, but felt this would be unnecessarily complex.
22044 Implementors are encouraged to meet the stronger requirement whenever possible. In practice,
22045 the weaker requirement is the same, except in the rare case when two signals arrive during a
22046 very short window. This reasoning also applies to a similar requirement for *sigprocmask()*.

22047 In 4.2 BSD, the SIGCONT signal can be sent to any descendant process regardless of user-ID
22048 security checks. This allows a job control shell to continue a job even if processes in the job have
22049 altered their user IDs (as in the *su* command). In keeping with the addition of the concept of
22050 sessions, similar functionality is provided by allowing the SIGCONT signal to be sent to any
22051 process in the same session regardless of user ID security checks. This is less restrictive than BSD
22052 in the sense that ancestor processes (in the same session) can now be the recipient. It is more
22053 restrictive than BSD in the sense that descendant processes that form new sessions are now
22054 subject to the user ID checks. A similar relaxation of security is not necessary for the other job
22055 control signals since those signals are typically sent by the terminal driver in recognition of
22056 special characters being typed; the terminal driver bypasses all security checks.

22057 In secure implementations, a process may be restricted from sending a signal to a process having
22058 a different security label. In order to prevent the existence or nonexistence of a process from
22059 being used as a covert channel, such processes should appear nonexistent to the sender; that is,
22060 [ESRCH] should be returned, rather than [EPERM], if *pid* refers only to such processes.

Existing implementations vary on the result of a *kill()* with *pid* indicating an inactive process (a terminated process that has not been waited for by its parent). Some indicate success on such a call (subject to permission checking), while others give an error of [ESRCH]. Since the definition of process lifetime in this volume of IEEE Std 1003.1-2001 covers inactive processes, the [ESRCH] error as described is inappropriate in this case. In particular, this means that an application cannot have a parent process check for termination of a particular child with *kill()*. (Usually this is done with the null signal; this can be done reliably with *waitpid()*.)

There is some belief that the name *kill()* is misleading, since the function is not always intended to cause process termination. However, the name is common to all historical implementations, and any change would be in conflict with the goal of minimal changes to existing application code.

22072 FUTURE DIRECTIONS

22073 None.

22074 SEE ALSO

22075 *getpid()*, *raise()*, *setsid()*, *sigaction()*, *sigqueue()*, the Base Definitions volume of
22076 IEEE Std 1003.1-2001, <signal.h>, <sys/types.h>

22077 CHANGE HISTORY

22078 First released in Issue 1. Derived from Issue 1 of the SVID.

22079 Issue 5

22080 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

22081 Issue 6

22082 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

22083 The following new requirements on POSIX implementations derive from alignment with the
22084 Single UNIX Specification:

- 22085 • In the DESCRIPTION, the second paragraph is reworded to indicate that the saved set-user-
22086 ID of the calling process is checked in place of its effective user ID. This is a FIPS
22087 requirement.
- 22088 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
22089 required for conforming implementations of previous POSIX specifications, it was not
22090 required for UNIX applications.
- 22091 • The behavior when *pid* is *-1* is now specified. It was previously explicitly unspecified in the
22092 POSIX.1-1988 standard.

22093 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

22094 **NAME**

22095 killpg — send a signal to a process group

22096 **SYNOPSIS**

22097 XSI #include <signal.h>

22098 int killpg(pid_t pgrp, int sig);

22099

22100 **DESCRIPTION**22101 The *killpg()* function shall send the signal specified by *sig* to the process group specified by *pgrp*.22102 If *pgrp* is greater than 1, *killpg(pgrp, sig)* shall be equivalent to *kill(-pgrp, sig)*. If *pgrp* is less than or
22103 equal to 1, the behavior of *killpg()* is undefined.22104 **RETURN VALUE**22105 Refer to *kill()*.22106 **ERRORS**22107 Refer to *kill()*.22108 **EXAMPLES**

22109 None.

22110 **APPLICATION USAGE**

22111 None.

22112 **RATIONALE**

22113 None.

22114 **FUTURE DIRECTIONS**

22115 None.

22116 **SEE ALSO**22117 *getpgid()*, *getpid()*, *kill()*, *raise()*, the Base Definitions volume of IEEE Std 1003.1-2001, <signal.h>22118 **CHANGE HISTORY**

22119 First released in Issue 4, Version 2.

22120 **Issue 5**

22121 Moved from X/OPEN UNIX extension to BASE.

22122 **NAME**

22123 l64a — convert a 32-bit integer to a radix-64 ASCII string

22124 **SYNOPSIS**

22125 xSI #include <stdlib.h>

22126 char *l64a(long value);

22127

22128 **DESCRIPTION**22129 Refer to *a64l()*.

22130 **NAME**

22131 labs, llabs — return a long integer absolute value

22132 **SYNOPSIS**

22133 #include <stdlib.h>

22134 long labs(long i);

22135 long long llabs(long long i);

22136 **DESCRIPTION**

22137 cx The functionality described on this reference page is aligned with the ISO C standard. Any
22138 conflict between the requirements described here and the ISO C standard is unintentional. This
22139 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

22140 The *labs()* function shall compute the absolute value of the **long** integer operand *i*. The *llabs()*
22141 function shall compute the absolute value of the **long long** integer operand *i*. If the result cannot
22142 be represented, the behavior is undefined.

22143 **RETURN VALUE**

22144 The *labs()* function shall return the absolute value of the **long** integer operand. The *labs()*
22145 function shall return the absolute value of the **long long** integer operand.

22146 **ERRORS**

22147 No errors are defined.

22148 **EXAMPLES**

22149 None.

22150 **APPLICATION USAGE**

22151 None.

22152 **RATIONALE**

22153 None.

22154 **FUTURE DIRECTIONS**

22155 None.

22156 **SEE ALSO**

22157 *abs()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

22158 **CHANGE HISTORY**

22159 First released in Issue 4. Derived from the ISO C standard.

22160 **Issue 6**

22161 The *llabs()* function is added for alignment with the ISO/IEC 9899:1999 standard.

22162 **NAME**

22163 lchown — change the owner and group of a symbolic link

22164 **SYNOPSIS**

22165 xSI #include <unistd.h>

22166 int lchown(const char *path, uid_t owner, gid_t group);

22167

22168 **DESCRIPTION**

22169 The *lchown()* function shall be equivalent to *chown()*, except in the case where the named file is a
 22170 symbolic link. In this case, *lchown()* shall change the ownership of the symbolic link file itself,
 22171 while *chown()* changes the ownership of the file or directory to which the symbolic link refers.

22172 **RETURN VALUE**

22173 Upon successful completion, *lchown()* shall return 0. Otherwise, it shall return -1 and set *errno* to
 22174 indicate an error.

22175 **ERRORS**22176 The *lchown()* function shall fail if:22177 [EACCES] Search permission is denied on a component of the path prefix of *path*.

22178 [EINVAL] The owner or group ID is not a value supported by the implementation.

22179 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 22180 argument.

22181 [ENAMETOOLONG]

22182 The length of a pathname exceeds {PATH_MAX} or a pathname component is
 22183 longer than {NAME_MAX}.

22184 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.22185 [ENOTDIR] A component of the path prefix of *path* is not a directory.

22186 [EOPNOTSUPP] The *path* argument names a symbolic link and the implementation does not
 22187 support setting the owner or group of a symbolic link.

22188 [EPERM] The effective user ID does not match the owner of the file and the process
 22189 does not have appropriate privileges.

22190 [EROFS] The file resides on a read-only file system.

22191 The *lchown()* function may fail if:

22192 [EIO] An I/O error occurred while reading or writing to the file system.

22193 [EINTR] A signal was caught during execution of the function.

22194 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 22195 resolution of the *path* argument.

22196 [ENAMETOOLONG]

22197 Pathname resolution of a symbolic link produced an intermediate result
 22198 whose length exceeds {PATH_MAX}.

22199 EXAMPLES**22200 Changing the Current Owner of a File**

22201 The following example shows how to change the ownership of the symbolic link named
22202 **/modules/pass1** to the user ID associated with “jones” and the group ID associated with “cnd”.

22203 The numeric value for the user ID is obtained by using the *getpwnam()* function. The numeric
22204 value for the group ID is obtained by using the *getgrnam()* function.

```
22205 #include <sys/types.h>
22206 #include <unistd.h>
22207 #include <pwd.h>
22208 #include <grp.h>

22209 struct passwd *pwd;
22210 struct group *grp;
22211 char          *path = "/modules/pass1";
22212 ...
22213 pwd = getpwnam("jones");
22214 grp = getgrnam("cnd");
22215 lchown(path, pwd->pw_uid, grp->gr_gid);
```

22216 APPLICATION USAGE

22217 On implementations which support symbolic links as directory entries rather than files, *lchown()*
22218 may fail.

22219 RATIONALE

22220 None.

22221 FUTURE DIRECTIONS

22222 None.

22223 SEE ALSO

22224 *chown()*, *symlink()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<unistd.h>**

22225 CHANGE HISTORY

22226 First released in Issue 4, Version 2.

22227 Issue 5

22228 Moved from X/OPEN UNIX extension to BASE.

22229 Issue 6

22230 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
22231 [ELOOP] error condition is added.

22232 The Open Group Base Resolution bwg2001-013 is applied, adding wording to the
22233 APPLICATION USAGE.

22234 **NAME**

22235 lcong48 — seed a uniformly distributed pseudo-random signed long integer generator

22236 **SYNOPSIS**

22237 xSI #include <stdlib.h>

22238 void lcong48(unsigned short param[7]);

22239

22240 **DESCRIPTION**

22241 Refer to *drand48()*.

22242 **NAME**

22243 ldexp, ldexpf, ldexpl — load exponent of a floating-point number

22244 **SYNOPSIS**

22245 #include <math.h>

22246 double ldexp(double x, int exp);

22247 float ldexpf(float x, int exp);

22248 long double ldexpl(long double x, int exp);

22249 **DESCRIPTION**

22250 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 22251 conflict between the requirements described here and the ISO C standard is unintentional. This
 22252 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

22253 These functions shall compute the quantity $x * 2^{exp}$.

22254 An application wishing to check for error situations should set *errno* to zero and call
 22255 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 22256 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 22257 zero, an error has occurred.

22258 **RETURN VALUE**22259 Upon successful completion, these functions shall return *x* multiplied by 2, raised to the power
22260 *exp*.

22261 If these functions would cause overflow, a range error shall occur and *ldexp*(), *ldexpf*(), and
 22262 *ldexpl*() shall return $\pm\text{HUGE_VAL}$, $\pm\text{HUGE_VALF}$, and $\pm\text{HUGE_VALL}$ (according to the sign of
 22263 *x*), respectively.

22264 If the correct value would cause underflow, and is not representable, a range error may occur,
 22265 MX and either 0.0 (if supported), or an implementation-defined value shall be returned.

22266 MX If *x* is NaN, a NaN shall be returned.22267 If *x* is ± 0 or $\pm\text{Inf}$, *x* shall be returned.22268 If *exp* is 0, *x* shall be returned.

22269 If the correct value would cause underflow, and is representable, a range error may occur and
 22270 the correct value shall be returned.

22271 **ERRORS**

22272 These functions shall fail if:

22273 Range Error The result overflows.

22274 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 22275 then *errno* shall be set to [ERANGE]. If the integer expression
 22276 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow
 22277 floating-point exception shall be raised.

22278 These functions may fail if:

22279 Range Error The result underflows.

22280 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 22281 then *errno* shall be set to [ERANGE]. If the integer expression
 22282 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
 22283 floating-point exception shall be raised.

22284 EXAMPLES

22285 None.

22286 APPLICATION USAGE

22287 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
22288 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

22289 RATIONALE

22290 None.

22291 FUTURE DIRECTIONS

22292 None.

22293 SEE ALSO

22294 *feclearexcept()*, *fetestexcept()*, *frexp()*, *isnan()*, the Base Definitions volume of
22295 IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
22296 <math.h>

22297 CHANGE HISTORY

22298 First released in Issue 1. Derived from Issue 1 of the SVID.

22299 Issue 5

22300 The DESCRIPTION is updated to indicate how an application should check for an error. This
22301 text was previously published in the APPLICATION USAGE section.

22302 Issue 6

22303 The *ldexpf()* and *ldexpl()* functions are added for alignment with the ISO/IEC 9899:1999
22304 standard.

22305 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
22306 revised to align with the ISO/IEC 9899:1999 standard.

22307 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
22308 marked.

22309 **NAME**

22310 ldiv, lldiv — compute quotient and remainder of a long division

22311 **SYNOPSIS**

22312 #include <stdlib.h>

22313 ldiv_t ldiv(long *numer*, long *denom*);22314 lldiv_t lldiv(long long *numer*, long long *denom*);22315 **DESCRIPTION**

22316 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 22317 conflict between the requirements described here and the ISO C standard is unintentional. This
 22318 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

22319 These functions shall compute the quotient and remainder of the division of the numerator
 22320 *numer* by the denominator *denom*. If the division is inexact, the resulting quotient is the **long**
 22321 integer (for the *ldiv()* function) or **long long** integer (for the *lldiv()* function) of lesser magnitude
 22322 that is the nearest to the algebraic quotient. If the result cannot be represented, the behavior is
 22323 undefined; otherwise, *quot* * *denom* + *rem* shall equal *numer*.

22324 **RETURN VALUE**

22325 The *ldiv()* function shall return a structure of type **ldiv_t**, comprising both the quotient and the
 22326 remainder. The structure shall include the following members, in any order:

22327 long quot; /* Quotient */
 22328 long rem; /* Remainder */

22329 The *lldiv()* function shall return a structure of type **lldiv_t**, comprising both the quotient and the
 22330 remainder. The structure shall include the following members, in any order:

22331 long long quot; /* Quotient */
 22332 long long rem; /* Remainder */

22333 **ERRORS**

22334 No errors are defined.

22335 **EXAMPLES**

22336 None.

22337 **APPLICATION USAGE**

22338 None.

22339 **RATIONALE**

22340 None.

22341 **FUTURE DIRECTIONS**

22342 None.

22343 **SEE ALSO**22344 *div()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>22345 **CHANGE HISTORY**

22346 First released in Issue 4. Derived from the ISO C standard.

22347 **Issue 6**22348 The *lldiv()* function is added for alignment with the ISO/IEC 9899:1999 standard.

22349 **NAME**

22350 lfind — find entry in a linear search table

22351 **SYNOPSIS**22352 XSI `#include <search.h>`

```
22353 void *lfind(const void *key, const void *base, size_t *nelp,  
22354           size_t width, int (*compar)(const void *, const void *));  
22355
```

22356 **DESCRIPTION**22357 Refer to *lsearch()*.

22358 **NAME**

22359 lgamma, lgammaf, lgammal — log gamma function

22360 **SYNOPSIS**

22361 #include <math.h>

22362 double lgamma(double x);

22363 float lgammaf(float x);

22364 long double lgammal(long double x);

22365 XSI extern int signgam;

22366

22367 **DESCRIPTION**

22368 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 22369 conflict between the requirements described here and the ISO C standard is unintentional. This
 22370 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

22371 These functions shall compute $\log_e |\Gamma(x)|$ where $\Gamma(x)$ is defined as $\int_0^{\infty} e^{-t} t^{x-1} dt$. The argument x
 22372 need not be a non-positive integer ($\Gamma(x)$ is defined over the reals, except the non-positive
 22373 integers).
 22374

22375 XSI The sign of $\Gamma(x)$ is returned in the external integer *signgam*.

22376 CX These functions need not be reentrant. A function that is not required to be reentrant is not
 22377 required to be thread-safe.

22378 An application wishing to check for error situations should set *errno* to zero and call
 22379 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 22380 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 22381 zero, an error has occurred.

22382 **RETURN VALUE**22383 Upon successful completion, these functions shall return the logarithmic gamma of x .

22384 If x is a non-positive integer, a pole error shall occur and *lgamma*(), *lgammaf*(), and *lgammal*()
 22385 shall return +HUGE_VAL, +HUGE_VALF, and +HUGE_VALL, respectively.

22386 If the correct value would cause overflow, a range error shall occur and *lgamma*(), *lgammaf*(),
 22387 and *lgammal*() shall return ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL (having the same
 22388 sign as the correct value), respectively.

22389 MX If x is NaN, a NaN shall be returned.22390 If x is 1 or 2, +0 shall be returned.22391 If x is ±Inf, +Inf shall be returned.22392 **ERRORS**

22393 These functions shall fail if:

22394 Pole Error The x argument is a negative integer or zero.

22395 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 22396 then *errno* shall be set to [ERANGE]. If the integer expression
 22397 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-
 22398 zero floating-point exception shall be raised.

22399 Range Error The result overflows.

22400 If the integer expression (`math_errhandling` & `MATH_ERRNO`) is non-zero,
22401 then *errno* shall be set to `[ERANGE]`. If the integer expression
22402 (`math_errhandling` & `MATH_ERREXCEPT`) is non-zero, then the overflow
22403 floating-point exception shall be raised.

22404 **EXAMPLES**

22405 None.

22406 **APPLICATION USAGE**

22407 On error, the expressions (`math_errhandling` & `MATH_ERRNO`) and (`math_errhandling` &
22408 `MATH_ERREXCEPT`) are independent of each other, but at least one of them must be non-zero.

22409 **RATIONALE**

22410 None.

22411 **FUTURE DIRECTIONS**

22412 None.

22413 **SEE ALSO**

22414 *exp()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-2001,
22415 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <**math.h**>

22416 **CHANGE HISTORY**

22417 First released in Issue 3.

22418 **Issue 5**

22419 The DESCRIPTION is updated to indicate how an application should check for an error. This
22420 text was previously published in the APPLICATION USAGE section.

22421 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

22422 **Issue 6**

22423 The *lgamma()* function is no longer marked as an extension.

22424 The *lgammaf()* and *lgammal()* functions are added for alignment with the ISO/IEC 9899:1999
22425 standard.

22426 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
22427 revised to align with the ISO/IEC 9899:1999 standard.

22428 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
22429 marked.

22430 XSI extensions are marked.

22431 **NAME**

22432 link — link to a file

22433 **SYNOPSIS**

22434 #include <unistd.h>

22435 int link(const char *path1, const char *path2);

22436 **DESCRIPTION**22437 The *link()* function shall create a new link (directory entry) for the existing file, *path1*.

22438 The *path1* argument points to a pathname naming an existing file. The *path2* argument points to
 22439 a pathname naming the new directory entry to be created. The *link()* function shall atomically
 22440 create a new link for the existing file and the link count of the file shall be incremented by one.

22441 If *path1* names a directory, *link()* shall fail unless the process has appropriate privileges and the
 22442 implementation supports using *link()* on directories.

22443 Upon successful completion, *link()* shall mark for update the *st_ctime* field of the file. Also, the
 22444 *st_ctime* and *st_mtime* fields of the directory that contains the new entry shall be marked for
 22445 update.

22446 If *link()* fails, no link shall be created and the link count of the file shall remain unchanged.

22447 The implementation may require that the calling process has permission to access the existing
 22448 file.

22449 **RETURN VALUE**

22450 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
 22451 indicate the error.

22452 **ERRORS**22453 The *link()* function shall fail if:

22454 [EACCES] A component of either path prefix denies search permission, or the requested
 22455 link requires writing in a directory that denies write permission, or the calling
 22456 process does not have permission to access the existing file and this is
 22457 required by the implementation.

22458 [EEXIST] The *path2* argument resolves to an existing file or refers to a symbolic link.

22459 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path1* or
 22460 *path2* argument.

22461 [EMLINK] The number of links to the file named by *path1* would exceed {LINK_MAX}.

22462 [ENAMETOOLONG]

22463 The length of the *path1* or *path2* argument exceeds {PATH_MAX} or a
 22464 pathname component is longer than {NAME_MAX}.

22465 [ENOENT] A component of either path prefix does not exist; the file named by *path1* does
 22466 not exist; or *path1* or *path2* points to an empty string.

22467 [ENOSPC] The directory to contain the link cannot be extended.

22468 [ENOTDIR] A component of either path prefix is not a directory.

22469 [EPERM] The file named by *path1* is a directory and either the calling process does not
 22470 have appropriate privileges or the implementation prohibits using *link()* on
 22471 directories.

22472 [EROFS] The requested link requires writing in a directory on a read-only file system.

22473 [EXDEV] The link named by *path2* and the file named by *path1* are on different file
22474 systems and the implementation does not support links between file systems.

22475 XSR [EXDEV] *path1* refers to a named STREAM.

22476 The *link()* function may fail if:

22477 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
22478 resolution of the *path1* or *path2* argument.

22479 [ENAMETOOLONG]
22480 As a result of encountering a symbolic link in resolution of the *path1* or *path2*
22481 argument, the length of the substituted pathname string exceeded
22482 {PATH_MAX}.

22483 EXAMPLES

22484 Creating a Link to a File

22485 The following example shows how to create a link to a file named **/home/cnd/mod1** by creating a
22486 new directory entry named **/modules/pass1**.

```
22487 #include <unistd.h>
22488 char *path1 = "/home/cnd/mod1";
22489 char *path2 = "/modules/pass1";
22490 int status;
22491 ...
22492 status = link (path1, path2);
```

22493 Creating a Link to a File Within a Program

22494 In the following program example, the *link()* function links the **/etc/passwd** file (defined as
22495 **PASSWDFILE**) to a file named **/etc/opasswd** (defined as **SAVEFILE**), which is used to save the
22496 current password file. Then, after removing the current password file (defined as
22497 **PASSWDFILE**), the new password file is saved as the current password file using the *link()*
22498 function again.

```
22499 #include <unistd.h>
22500 #define LOCKFILE "/etc/ptmp"
22501 #define PASSWDFILE "/etc/passwd"
22502 #define SAVEFILE "/etc/opasswd"
22503 ...
22504 /* Save current password file */
22505 link (PASSWDFILE, SAVEFILE);
22506
22507 /* Remove current password file. */
22508 unlink (PASSWDFILE);
22509
22508 /* Save new password file as current password file. */
22509 link (LOCKFILE, PASSWDFILE);
```


22510 APPLICATION USAGE

22511 Some implementations do allow links between file systems.

22512 RATIONALE

22513 Linking to a directory is restricted to the superuser in most historical implementations because
22514 this capability may produce loops in the file hierarchy or otherwise corrupt the file system. This
22515 volume of IEEE Std 1003.1-2001 continues that philosophy by prohibiting *link()* and *unlink()*
22516 from doing this. Other functions could do it if the implementor designed such an extension.

22517 Some historical implementations allow linking of files on different file systems. Wording was
22518 added to explicitly allow this optional behavior.

22519 The exception for cross-file system links is intended to apply only to links that are
22520 programmatically indistinguishable from “hard” links.

22521 FUTURE DIRECTIONS

22522 None.

22523 SEE ALSO

22524 *symlink()*, *unlink()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**unistd.h**>

22525 CHANGE HISTORY

22526 First released in Issue 1. Derived from Issue 1 of the SVID.

22527 Issue 6

22528 The following new requirements on POSIX implementations derive from alignment with the
22529 Single UNIX Specification:

- 22530 • The [ELOOP] mandatory error condition is added.
- 22531 • A second [ENAMETOOLONG] is added as an optional error condition.

22532 The following changes were made to align with the IEEE P1003.1a draft standard:

- 22533 • An explanation is added of the action when *path2* refers to a symbolic link.
- 22534 • The [ELOOP] optional error condition is added.

22535 **NAME**22536 lio_listio — list directed I/O (**REALTIME**)22537 **SYNOPSIS**

22538 AIO #include <aio.h>

```
22539       int lio_listio(int mode, struct aiocb *restrict const list[restrict],
22540           int nent, struct sigevent *restrict sig);
```

22541

22542 **DESCRIPTION**22543 The *lio_listio()* function shall initiate a list of I/O requests with a single function call.

22544 The *mode* argument takes one of the values LIO_WAIT or LIO_NOWAIT declared in <aio.h> and
 22545 determines whether the function returns when the I/O operations have been completed, or as
 22546 soon as the operations have been queued. If the *mode* argument is LIO_WAIT, the function shall
 22547 wait until all I/O is complete and the *sig* argument shall be ignored.

22548 If the *mode* argument is LIO_NOWAIT, the function shall return immediately, and asynchronous
 22549 notification shall occur, according to the *sig* argument, when all the I/O operations complete. If
 22550 *sig* is NULL, then no asynchronous notification shall occur. If *sig* is not NULL, asynchronous
 22551 notification occurs as specified in Section 2.4.1 (on page 28) when all the requests in *list* have
 22552 completed.

22553 The I/O requests enumerated by *list* are submitted in an unspecified order.

22554 The *list* argument is an array of pointers to **aiocb** structures. The array contains *nent* elements.
 22555 The array may contain NULL elements, which shall be ignored.

22556 The *aio_lio_opcode* field of each **aiocb** structure specifies the operation to be performed. The
 22557 supported operations are LIO_READ, LIO_WRITE, and LIO_NOP; these symbols are defined in
 22558 <aio.h>. The LIO_NOP operation causes the list entry to be ignored. If the *aio_lio_opcode*
 22559 element is equal to LIO_READ, then an I/O operation is submitted as if by a call to *aio_read()*
 22560 with the *aiocbp* equal to the address of the **aiocb** structure. If the *aio_lio_opcode* element is equal
 22561 to LIO_WRITE, then an I/O operation is submitted as if by a call to *aio_write()* with the *aiocbp*
 22562 equal to the address of the **aiocb** structure.

22563 The *aio_fildes* member specifies the file descriptor on which the operation is to be performed.22564 The *aio_buf* member specifies the address of the buffer to or from which the data is transferred.22565 The *aio_nbytes* member specifies the number of bytes of data to be transferred.

22566 The members of the **aiocb** structure further describe the I/O operation to be performed, in a
 22567 manner identical to that of the corresponding **aiocb** structure when used by the *aio_read()* and
 22568 *aio_write()* functions.

22569 The *nent* argument specifies how many elements are members of the list; that is, the length of the
 22570 array.

22571 The behavior of this function is altered according to the definitions of synchronized I/O data
 22572 integrity completion and synchronized I/O file integrity completion if synchronized I/O is
 22573 enabled on the file associated with *aio_fildes*.

22574 For regular files, no data transfer shall occur past the offset maximum established in the open
 22575 file description associated with *aiocbp->aio_fildes*.

22576 RETURN VALUE

22577 If the *mode* argument has the value LIO_NOWAIT, the *lio_listio()* function shall return the value
 22578 zero if the I/O operations are successfully queued; otherwise, the function shall return the value
 22579 -1 and set *errno* to indicate the error.

22580 If the *mode* argument has the value LIO_WAIT, the *lio_listio()* function shall return the value
 22581 zero when all the indicated I/O has completed successfully. Otherwise, *lio_listio()* shall return a
 22582 value of -1 and set *errno* to indicate the error.

22583 In either case, the return value only indicates the success or failure of the *lio_listio()* call itself,
 22584 not the status of the individual I/O requests. In some cases one or more of the I/O requests
 22585 contained in the list may fail. Failure of an individual request does not prevent completion of
 22586 any other individual request. To determine the outcome of each I/O request, the application
 22587 shall examine the error status associated with each **aiocb** control block. The error statuses so
 22588 returned are identical to those returned as the result of an *aio_read()* or *aio_write()* function.

22589 ERRORS

22590 The *lio_listio()* function shall fail if:

22591 [EAGAIN] The resources necessary to queue all the I/O requests were not available. The
 22592 application may check the error status for each **aiocb** to determine the
 22593 individual request(s) that failed.

22594 [EAGAIN] The number of entries indicated by *nent* would cause the system-wide limit
 22595 {AIO_MAX} to be exceeded.

22596 [EINVAL] The *mode* argument is not a proper value, or the value of *nent* was greater than
 22597 {AIO_LISTIO_MAX}.

22598 [EINTR] A signal was delivered while waiting for all I/O requests to complete during
 22599 an LIO_WAIT operation. Note that, since each I/O operation invoked by
 22600 *lio_listio()* may possibly provoke a signal when it completes, this error return
 22601 may be caused by the completion of one (or more) of the very I/O operations
 22602 being awaited. Outstanding I/O requests are not canceled, and the application
 22603 shall examine each list element to determine whether the request was
 22604 initiated, canceled, or completed.

22605 [EIO] One or more of the individual I/O operations failed. The application may
 22606 check the error status for each **aiocb** structure to determine the individual
 22607 request(s) that failed.

22608 In addition to the errors returned by the *lio_listio()* function, if the *lio_listio()* function succeeds
 22609 or fails with errors of [EAGAIN], [EINTR], or [EIO], then some of the I/O specified by the list
 22610 may have been initiated. If the *lio_listio()* function fails with an error code other than [EAGAIN],
 22611 [EINTR], or [EIO], no operations from the list shall have been initiated. The I/O operation
 22612 indicated by each list element can encounter errors specific to the individual read or write
 22613 function being performed. In this event, the error status for each **aiocb** control block contains the
 22614 associated error code. The error codes that can be set are the same as would be set by a *read()* or
 22615 *write()* function, with the following additional error codes possible:

22616 [EAGAIN] The requested I/O operation was not queued due to resource limitations.

22617 [ECANCELED] The requested I/O was canceled before the I/O completed due to an explicit
 22618 *aio_cancel()* request.

22619 [EFBIG] The *aiocbp->aio_lio_opcode* is LIO_WRITE, the file is a regular file,
 22620 *aiocbp->aio_nbytes* is greater than 0, and the *aiocbp->aio_offset* is greater than or
 22621 equal to the offset maximum in the open file description associated with

22622 *aiocbp->aio_fildes*.

22623 [EINPROGRESS] The requested I/O is in progress.

22624 [EOVERFLOW] The *aiocbp->aio_lio_opcode* is LIO_READ, the file is a regular file,
 22625 *aiocbp->aio_nbytes* is greater than 0, and the *aiocbp->aio_offset* is before the
 22626 end-of-file and is greater than or equal to the offset maximum in the open file
 22627 description associated with *aiocbp->aio_fildes*.

22628 EXAMPLES

22629 None.

22630 APPLICATION USAGE

22631 None.

22632 RATIONALE

22633 Although it may appear that there are inconsistencies in the specified circumstances for error
 22634 codes, the [EIO] error condition applies when any circumstance relating to an individual
 22635 operation makes that operation fail. This might be due to a badly formulated request (for
 22636 example, the *aio_lio_opcode* field is invalid, and *aio_error()* returns [EINVAL]) or might arise from
 22637 application behavior (for example, the file descriptor is closed before the operation is initiated,
 22638 and *aio_error()* returns [EBADF]).

22639 The limitation on the set of error codes returned when operations from the list shall have been
 22640 initiated enables applications to know when operations have been started and whether
 22641 *aio_error()* is valid for a specific operation.

22642 FUTURE DIRECTIONS

22643 None.

22644 SEE ALSO

22645 *aio_read()*, *aio_write()*, *aio_error()*, *aio_return()*, *aio_cancel()*, *close()*, *exec*, *exit()*, *fork()*, *lseek()*,
 22646 *read()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**aio.h**>

22647 CHANGE HISTORY

22648 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

22649 Large File Summit extensions are added.

22650 Issue 6

22651 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 22652 implementation does not support the Asynchronous Input and Output option.

22653 The *lio_listio()* function is marked as part of the Asynchronous Input and Output option.

22654 The following new requirements on POSIX implementations derive from alignment with the
 22655 Single UNIX Specification:

22656 • In the DESCRIPTION, text is added to indicate that for regular files no data transfer occurs
 22657 past the offset maximum established in the open file description associated with
 22658 *aiocbp->aio_fildes*. This change is to support large files.

22659 • The [EBIG] and [EOVERFLOW] error conditions are defined. This change is to support large
 22660 files.

22661 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

22662 The **restrict** keyword is added to the *lio_listio()* prototype for alignment with the
 22663 ISO/IEC 9899:1999 standard.

22664 **NAME**

22665 listen — listen for socket connections and limit the queue of incoming connections

22666 **SYNOPSIS**

22667 #include <sys/socket.h>

22668 int listen(int *socket*, int *backlog*);

22669 **DESCRIPTION**

22670 The *listen()* function shall mark a connection-mode socket, specified by the *socket* argument, as
 22671 accepting connections.

22672 The *backlog* argument provides a hint to the implementation which the implementation shall use
 22673 to limit the number of outstanding connections in the socket's listen queue. Implementations
 22674 may impose a limit on *backlog* and silently reduce the specified value. Normally, a larger *backlog*
 22675 argument value shall result in a larger or equal length of the listen queue. Implementations shall
 22676 support values of *backlog* up to SOMAXCONN, defined in <sys/socket.h>.

22677 The implementation may include incomplete connections in its listen queue. The limits on the
 22678 number of incomplete connections and completed connections queued may be different.

22679 The implementation may have an upper limit on the length of the listen queue—either global or
 22680 per accepting socket. If *backlog* exceeds this limit, the length of the listen queue is set to the limit.

22681 If *listen()* is called with a *backlog* argument value that is less than 0, the function behaves as if it
 22682 had been called with a *backlog* argument value of 0.

22683 A *backlog* argument of 0 may allow the socket to accept connections, in which case the length of
 22684 the listen queue may be set to an implementation-defined minimum value.

22685 The socket in use may require the process to have appropriate privileges to use the *listen()*
 22686 function.

22687 **RETURN VALUE**

22688 Upon successful completions, *listen()* shall return 0; otherwise, -1 shall be returned and *errno* set
 22689 to indicate the error.

22690 **ERRORS**

22691 The *listen()* function shall fail if:

22692 [EBADF] The *socket* argument is not a valid file descriptor.

22693 [EDESTADDRREQ]

22694 The socket is not bound to a local address, and the protocol does not support
 22695 listening on an unbound socket.

22696 [EINVAL] The *socket* is already connected.

22697 [ENOTSOCK] The *socket* argument does not refer to a socket.

22698 [EOPNOTSUPP] The socket protocol does not support *listen()*.

22699 The *listen()* function may fail if:

22700 [EACCES] The calling process does not have the appropriate privileges.

22701 [EINVAL] The *socket* has been shut down.

22702 [ENOBUFS] Insufficient resources are available in the system to complete the call.

22703 **EXAMPLES**

22704 None.

22705 **APPLICATION USAGE**

22706 None.

22707 **RATIONALE**

22708 None.

22709 **FUTURE DIRECTIONS**

22710 None.

22711 **SEE ALSO**22712 *accept()*, *connect()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>22713 **CHANGE HISTORY**

22714 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

22715 The DESCRIPTION is updated to describe the relationship of SOMAXCONN and the *backlog*
22716 argument.

22717 NAME

22718 llabs — return a long integer absolute value

22719 SYNOPSIS

22720 #include <stdlib.h>

22721 long long llabs(long long *i*);

22722 DESCRIPTION

22723 Refer to *labs()*.

22724 **NAME**22725 `lldiv` — compute quotient and remainder of a long division22726 **SYNOPSIS**22727 `#include <stdlib.h>`22728 `lldiv_t lldiv(long long numer, long long denom);`22729 **DESCRIPTION**22730 Refer to *ldiv()*.

22731 NAME

22732 llrint, llrintf, llrintl, — round to the nearest integer value using current rounding direction

22733 SYNOPSIS

22734 #include <math.h>

22735 long long llrint(double x);

22736 long long llrintf(float x);

22737 long long llrintl(long double x);

22738 DESCRIPTION

22739 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 22740 conflict between the requirements described here and the ISO C standard is unintentional. This
 22741 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

22742 These functions shall round their argument to the nearest integer value, rounding according to
 22743 the current rounding direction.

22744 An application wishing to check for error situations should set *errno* to zero and call
 22745 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 22746 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 22747 zero, an error has occurred.

22748 RETURN VALUE

22749 Upon successful completion, these functions shall return the rounded integer value.

22750 MX If *x* is NaN, a domain error shall occur, and an unspecified value is returned.

22751 If *x* is +Inf, a domain error shall occur and an unspecified value is returned.

22752 If *x* is -Inf, a domain error shall occur and an unspecified value is returned.

22753 If the correct value is positive and too large to represent as a **long long**, a domain error shall
 22754 occur and an unspecified value is returned.

22755 If the correct value is negative and too large to represent as a **long long**, a domain error shall
 22756 occur and an unspecified value is returned.

22757 ERRORS

22758 These functions shall fail if:

22759 MX	Domain Error	The <i>x</i> argument is NaN or \pm Inf, or the correct value is not representable as an integer.
----------	--------------	---

22761	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then <i>errno</i> shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.
-------	--

22765 EXAMPLES

22766 None.

22767 APPLICATION USAGE

22768 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
 22769 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

22770 RATIONALE

22771 These functions provide floating-to-integer conversions. They round according to the current
 22772 rounding direction. If the rounded value is outside the range of the return type, the numeric
 22773 result is unspecified and the invalid floating-point exception is raised. When they raise no other
 22774 floating-point exception and the result differs from the argument, they raise the inexact

22775 floating-point exception.

22776 **FUTURE DIRECTIONS**

22777 None.

22778 **SEE ALSO**

22779 *feclearexcept()*, *fetestexcept()*, *lrint()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section
22780 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

22781 **CHANGE HISTORY**

22782 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

22783 NAME

22784 llround, llroundf, llroundl, — round to nearest integer value

22785 SYNOPSIS

22786 #include <math.h>

22787 long long llround(double x);

22788 long long llroundf(float x);

22789 long long llroundl(long double x);

22790 DESCRIPTION

22791 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 22792 conflict between the requirements described here and the ISO C standard is unintentional. This
 22793 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

22794 These functions shall round their argument to the nearest integer value, rounding halfway cases
 22795 away from zero, regardless of the current rounding direction.

22796 An application wishing to check for error situations should set *errno* to zero and call
 22797 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 22798 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 22799 zero, an error has occurred.

22800 RETURN VALUE

22801 Upon successful completion, these functions shall return the rounded integer value.

22802 MX If *x* is NaN, a domain error shall occur, and an unspecified value is returned.

22803 If *x* is +Inf, a domain error shall occur and an unspecified value is returned.

22804 If *x* is -Inf, a domain error shall occur and an unspecified value is returned.

22805 If the correct value is positive and too large to represent as a **long long**, a domain error shall
 22806 occur and an unspecified value is returned.

22807 If the correct value is negative and too large to represent as a **long long**, a domain error shall
 22808 occur and an unspecified value is returned.

22809 ERRORS

22810 These functions shall fail if:

22811 MX	Domain Error	The <i>x</i> argument is NaN or \pm Inf, or the correct value is not representable as an integer.
----------	--------------	---

22813	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
22814	then <i>errno</i> shall be set to [EDOM]. If the integer expression (math_errhandling
22815	& MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
22816	shall be raised.

22817 EXAMPLES

22818 None.

22819 APPLICATION USAGE

22820 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
 22821 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

22822 RATIONALE

22823 These functions differ from the *llrint*() functions in that the default rounding direction for the
 22824 *llround*() functions round halfway cases away from zero and need not raise the inexact floating-
 22825 point exception for non-integer arguments that round to within the range of the return type.

22826 **FUTURE DIRECTIONS**

22827 None.

22828 **SEE ALSO**

22829 *feclearexcept()*, *fetestexcept()*, *lround()*, the Base Definitions volume of IEEE Std 1003.1-2001,
22830 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

22831 **CHANGE HISTORY**

22832 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

22833 NAME

22834 localeconv — return locale-specific information

22835 SYNOPSIS

22836 #include <locale.h>

22837 struct lconv *localeconv(void);

22838 DESCRIPTION

22839 cx The functionality described on this reference page is aligned with the ISO C standard. Any
 22840 conflict between the requirements described here and the ISO C standard is unintentional. This
 22841 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

22842 The *localeconv()* function shall set the components of an object with the type **struct lconv** with
 22843 the values appropriate for the formatting of numeric quantities (monetary and otherwise)
 22844 according to the rules of the current locale.

22845 The members of the structure with type **char *** are pointers to strings, any of which (except
 22846 **decimal_point**) can point to " ", to indicate that the value is not available in the current locale or
 22847 is of zero length. The members with type **char** are non-negative numbers, any of which can be
 22848 {CHAR_MAX} to indicate that the value is not available in the current locale.

22849 The members include the following:

22850 **char *decimal_point**

22851 The radix character used to format non-monetary quantities.

22852 **char *thousands_sep**

22853 The character used to separate groups of digits before the decimal-point character in
 22854 formatted non-monetary quantities.

22855 **char *grouping**

22856 A string whose elements taken as one-byte integer values indicate the size of each group of
 22857 digits in formatted non-monetary quantities.

22858 **char *int_curr_symbol**

22859 The international currency symbol applicable to the current locale. The first three
 22860 characters contain the alphabetic international currency symbol in accordance with those
 22861 specified in the ISO 4217:1995 standard. The fourth character (immediately preceding the
 22862 null byte) is the character used to separate the international currency symbol from the
 22863 monetary quantity.

22864 **char *currency_symbol**

22865 The local currency symbol applicable to the current locale.

22866 **char *mon_decimal_point**

22867 The radix character used to format monetary quantities.

22868 **char *mon_thousands_sep**

22869 The separator for groups of digits before the decimal-point in formatted monetary
 22870 quantities.

22871 **char *mon_grouping**

22872 A string whose elements taken as one-byte integer values indicate the size of each group of
 22873 digits in formatted monetary quantities.

22874 **char *positive_sign**

22875 The string used to indicate a non-negative valued formatted monetary quantity.

22876 **char *negative_sign**
 22877 The string used to indicate a negative valued formatted monetary quantity.

22878 **char int_frac_digits**
 22879 The number of fractional digits (those after the decimal-point) to be displayed in an
 22880 internationally formatted monetary quantity.

22881 **char frac_digits**
 22882 The number of fractional digits (those after the decimal-point) to be displayed in a
 22883 formatted monetary quantity.

22884 **char p_cs_precedes**
 22885 Set to 1 if the **currency_symbol** or **int_curr_symbol** precedes the value for a non-negative
 22886 formatted monetary quantity. Set to 0 if the symbol succeeds the value.

22887 **char p_sep_by_space**
 22888 Set to 0 if no space separates the **currency_symbol** or **int_curr_symbol** from the value for a
 22889 non-negative formatted monetary quantity. Set to 1 if a space separates the symbol from the
 22890 value; and set to 2 if a space separates the symbol and the sign string, if adjacent. XSI

22891 **char n_cs_precedes**
 22892 Set to 1 if the **currency_symbol** or **int_curr_symbol** precedes the value for a negative
 22893 formatted monetary quantity. Set to 0 if the symbol succeeds the value.

22894 **char n_sep_by_space**
 22895 Set to 0 if no space separates the **currency_symbol** or **int_curr_symbol** from the value for a
 22896 negative formatted monetary quantity. Set to 1 if a space separates the symbol from the
 22897 value; and set to 2 if a space separates the symbol and the sign string, if adjacent. XSI

22898 **char p_sign_posn**
 22899 Set to a value indicating the positioning of the **positive_sign** for a non-negative formatted
 22900 monetary quantity.

22901 **char n_sign_posn**
 22902 Set to a value indicating the positioning of the **negative_sign** for a negative formatted
 22903 monetary quantity.

22904 **char int_p_cs_precedes**
 22905 Set to 1 or 0 if the **int_curr_symbol** respectively precedes or succeeds the value for a non-
 22906 negative internationally formatted monetary quantity.

22907 **char int_n_cs_precedes**
 22908 Set to 1 or 0 if the **int_curr_symbol** respectively precedes or succeeds the value for a
 22909 negative internationally formatted monetary quantity.

22910 **char int_p_sep_by_space**
 22911 Set to a value indicating the separation of the **int_curr_symbol**, the sign string, and the
 22912 value for a non-negative internationally formatted monetary quantity.

22913 **char int_n_sep_by_space**
 22914 Set to a value indicating the separation of the **int_curr_symbol**, the sign string, and the
 22915 value for a negative internationally formatted monetary quantity.

22916 **char int_p_sign_posn**
 22917 Set to a value indicating the positioning of the **positive_sign** for a non-negative
 22918 internationally formatted monetary quantity.

22919 **char int_n_sign_posn**
 22920 Set to a value indicating the positioning of the **negative_sign** for a negative internationally

22921 formatted monetary quantity.

22922 The elements of **grouping** and **mon_grouping** are interpreted according to the following:

22923 {CHAR_MAX} No further grouping is to be performed.

22924 0 The previous element is to be repeatedly used for the remainder of the digits.

22925 *other* The integer value is the number of digits that comprise the current group. The
22926 next element is examined to determine the size of the next group of digits
22927 before the current group.

22928 The values of **p_sep_by_space**, **n_sep_by_space**, **int_p_sep_by_space**, and **int_n_sep_by_space**
22929 are interpreted according to the following:

22930 0 No space separates the currency symbol and value.

22931 1 If the currency symbol and sign string are adjacent, a space separates them from the value;
22932 otherwise, a space separates the currency symbol from the value.

22933 2 If the currency symbol and sign string are adjacent, a space separates them; otherwise, a
22934 space separates the sign string from the value.

22935 For **int_p_sep_by_space** and **int_n_sep_by_space**, the fourth character of **int_curr_symbol** is
22936 used instead of a space.

22937 The values of **p_sign_posn**, **n_sign_posn**, **int_p_sign_posn**, and **int_n_sign_posn** are
22938 interpreted according to the following:

22939 0 Parentheses surround the quantity and **currency_symbol** or **int_curr_symbol**.

22940 1 The sign string precedes the quantity and **currency_symbol** or **int_curr_symbol**.

22941 2 The sign string succeeds the quantity and **currency_symbol** or **int_curr_symbol**.

22942 3 The sign string immediately precedes the **currency_symbol** or **int_curr_symbol**.

22943 4 The sign string immediately succeeds the **currency_symbol** or **int_curr_symbol**.

22944 The implementation shall behave as if no function in this volume of IEEE Std 1003.1-2001 calls
22945 *localeconv()*.

22946 CX The *localeconv()* function need not be reentrant. A function that is not required to be reentrant is
22947 not required to be thread-safe.

22948 RETURN VALUE

22949 The *localeconv()* function shall return a pointer to the filled-in object. The application shall not
22950 modify the structure pointed to by the return value which may be overwritten by a subsequent
22951 call to *localeconv()*. In addition, calls to *setlocale()* with the categories *LC_ALL*, *LC_MONETARY*,
22952 or *LC_NUMERIC* may overwrite the contents of the structure.

22953 ERRORS

22954 No errors are defined.

22955 **EXAMPLES**

22956 None.

22957 **APPLICATION USAGE**

22958 The following table illustrates the rules which may be used by four countries to format monetary
 22959 quantities.

Country	Positive Format	Negative Format	International Format
Italy	L.1.230	–L.1.230	ITL.1.230
Netherlands	F 1.234,56	F –1.234,56	NLG 1.234,56
Norway	kr1.234,56	kr1.234,56–	NOK 1.234,56
Switzerland	SFrs.1,234.56	SFrs.1,234.56C	CHF 1,234.56

22965 For these four countries, the respective values for the monetary members of the structure
 22966 returned by *localeconv()* are:

	Italy	Netherlands	Norway	Switzerland
int_curr_symbol	"ITL. "	"NLG "	"NOK "	"CHF "
currency_symbol	"L. "	"F "	"kr "	"SFrs. "
mon_decimal_point	" "	" , "	" , "	" . "
mon_thousands_sep	" . "	" . "	" . "	" / "
mon_grouping	"\3 "	"\3 "	"\3 "	"\3 "
positive_sign	" "	" "	" "	" "
negative_sign	" – "	" – "	" – "	" C "
int_frac_digits	0	2	2	2
frac_digits	0	2	2	2
p_cs_precedes	1	1	1	1
p_sep_by_space	0	1	0	0
n_cs_precedes	1	1	1	1
n_sep_by_space	0	1	0	0
p_sign_posn	1	1	1	1
n_sign_posn	1	4	2	2
int_p_cs_precedes	1	1	1	1
int_n_cs_precedes	1	1	1	1
int_p_sep_by_space	0	0	0	0
int_n_sep_by_space	0	0	0	0
int_p_sign_posn	1	1	1	1
int_n_sign_posn	1	4	4	2

22989 **RATIONALE**

22990 None.

22991 **FUTURE DIRECTIONS**

22992 None.

22993 **SEE ALSO**

22994 *isalpha()*, *isascii()*, *nl_langinfo()*, *printf()*, *scanf()*, *setlocale()*, *strcat()*, *strchr()*, *strcmp()*, *strcoll()*,
 22995 *strcpy()*, *strftime()*, *strlen()*, *strpbrk()*, *strspn()*, *strtok()*, *strxfrm()*, *strtod()*, the Base Definitions
 22996 volume of IEEE Std 1003.1-2001, <langinfo.h>, <locale.h>

22997 **CHANGE HISTORY**

22998 First released in Issue 4. Derived from the ANSI C standard.

22999 Issue 6

23000 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

23001 The RETURN VALUE section is rewritten to avoid use of the term “must”.

23002 This reference page is updated for alignment with the ISO/IEC 9899: 1999 standard.

23003 ISO/IEC 9899: 1999 standard, Technical Corrigendum No. 1 is incorporated.

23004 **NAME**

23005 localtime, localtime_r — convert a time value to a broken-down local time

23006 **SYNOPSIS**

23007 #include <time.h>

23008 struct tm *localtime(const time_t *timer);

23009 TSF struct tm *localtime_r(const time_t *restrict timer,

23010 struct tm *restrict result);

23011

23012 **DESCRIPTION**23013 CX For *localtime()*: The functionality described on this reference page is aligned with the ISO C
23014 standard. Any conflict between the requirements described here and the ISO C standard is
23015 unintentional. This volume of IEEE Std 1003.1-2001 defers to the ISO C standard.23016 The *localtime()* function shall convert the time in seconds since the Epoch pointed to by *timer*
23017 into a broken-down time, expressed as a local time. The function corrects for the timezone and
23018 CX any seasonal time adjustments. Local timezone information is used as though *localtime()* calls
23019 *tzset()*.23020 The relationship between a time in seconds since the Epoch used as an argument to *localtime()*
23021 and the **tm** structure (defined in the <time.h> header) is that the result shall be as specified in the
23022 expression given in the definition of seconds since the Epoch (see the Base Definitions volume of
23023 IEEE Std 1003.1-2001, Section 4.14, Seconds Since the Epoch) corrected for timezone and any
23024 seasonal time adjustments, where the names in the structure and in the expression correspond.23025 TSF The same relationship shall apply for *localtime_r()*.23026 CX The *localtime()* function need not be reentrant. A function that is not required to be reentrant is
23027 not required to be thread-safe.23028 The *asctime()*, *ctime()*, *gmtime()*, and *localtime()* functions shall return values in one of two static
23029 objects: a broken-down time structure and an array of type **char**. Execution of any of the
23030 functions may overwrite the information returned in either of these objects by any of the other
23031 functions.23032 TSF The *localtime_r()* function shall convert the time in seconds since the Epoch pointed to by *timer*
23033 into a broken-down time stored in the structure to which *result* points. The *localtime_r()* function
23034 shall also return a pointer to that same structure.23035 Unlike *localtime()*, the reentrant version is not required to set *tzname*.23036 **RETURN VALUE**23037 The *localtime()* function shall return a pointer to the broken-down time structure.23038 TSF Upon successful completion, *localtime_r()* shall return a pointer to the structure pointed to by
23039 the argument *result*.23040 **ERRORS**

23041 No errors are defined.

23042 **EXAMPLES**23043 **Getting the Local Date and Time**

23044 The following example uses the *time()* function to calculate the time elapsed, in seconds, since
 23045 January 1, 1970 0:00 UTC (the Epoch), *localtime()* to convert that value to a broken-down time,
 23046 and *asctime()* to convert the broken-down time values into a printable string.

```
23047 #include <stdio.h>
23048 #include <time.h>

23049 int main(void)
23050 {
23051     time_t result;

23052     result = time(NULL);
23053     printf("%s%ju secs since the Epoch\n",
23054           asctime(localtime(&result)),
23055           (uintmax_t)result);
23056     return(0);
23057 }
```

23058 This example writes the current time to *stdout* in a form like this:

```
23059 Wed Jun 26 10:32:15 1996
23060 835810335 secs since the Epoch
```

23061 **Getting the Modification Time for a File**

23062 The following example gets the modification time for a file. The *localtime()* function converts the
 23063 **time_t** value of the last modification date, obtained by a previous call to *stat()*, into a **tm**
 23064 structure that contains the year, month, day, and so on.

```
23065 #include <time.h>
23066 ...
23067 struct stat statbuf;
23068 ...
23069 tm = localtime(&statbuf.st_mtime);
23070 ...
```

23071 **Timing an Event**

23072 The following example gets the current time, converts it to a string using *localtime()* and
 23073 *asctime()*, and prints it to standard output using *fputs()*. It then prints the number of minutes to
 23074 an event being timed.

```
23075 #include <time.h>
23076 #include <stdio.h>
23077 ...
23078 time_t now;
23079 int minutes_to_event;
23080 ...
23081 time(&now);
23082 printf("The time is ");
23083 fputs(asctime(localtime(&now)), stdout);
23084 printf("There are still %d minutes to the event.\n",
```


23085 minutes_to_event);
23086 ...

23087 **APPLICATION USAGE**

23088 The *localtime_r()* function is thread-safe and returns values in a user-supplied buffer instead of
23089 possibly using a static data area that may be overwritten by each call.

23090 **RATIONALE**

23091 None.

23092 **FUTURE DIRECTIONS**

23093 None.

23094 **SEE ALSO**

23095 *asctime()*, *clock()*, *ctime()*, *difftime()*, *getdate()*, *gmtime()*, *mktime()*, *strftime()*, *strptime()*, *time()*,
23096 *utime()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**time.h**>

23097 **CHANGE HISTORY**

23098 First released in Issue 1. Derived from Issue 1 of the SVID.

23099 **Issue 5**

23100 A note indicating that the *localtime()* function need not be reentrant is added to the
23101 DESCRIPTION.

23102 The *localtime_r()* function is included for alignment with the POSIX Threads Extension.

23103 **Issue 6**

23104 The *localtime_r()* function is marked as part of the Thread-Safe Functions option.

23105 Extensions beyond the ISO C standard are marked.

23106 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
23107 its avoidance of possibly using a static data area.

23108 The **restrict** keyword is added to the *localtime_r()* prototype for alignment with the
23109 ISO/IEC 9899:1999 standard.

23110 Examples are added.

23111 **NAME**

23112 lockf — record locking on files

23113 **SYNOPSIS**23114 XSI `#include <unistd.h>`23115 `int lockf(int fildes, int function, off_t size);`

23116

23117 **DESCRIPTION**

23118 The *lockf()* function shall lock sections of a file with advisory-mode locks. Calls to *lockf()* from
 23119 other threads which attempt to lock the locked file section shall either return an error value or
 23120 block until the section becomes unlocked. All the locks for a process are removed when the
 23121 process terminates. Record locking with *lockf()* shall be supported for regular files and may be
 23122 supported for other files.

23123 The *fildes* argument is an open file descriptor. To establish a lock with this function, the file
 23124 descriptor shall be opened with write-only permission (O_WRONLY) or with read/write
 23125 permission (O_RDWR).

23126 The *function* argument is a control value which specifies the action to be taken. The permissible
 23127 values for *function* are defined in <unistd.h> as follows:

23128

23129

23130

23131

23132

23133

Function	Description
F_ULOCK	Unlock locked sections.
F_LOCK	Lock a section for exclusive use.
F_TLOCK	Test and lock a section for exclusive use.
F_TEST	Test a section for locks by other processes.

23134 F_TEST shall detect if a lock by another process is present on the specified section.

23135 F_LOCK and F_TLOCK shall both lock a section of a file if the section is available.

23136 F_ULOCK shall remove locks from a section of the file.

23137 The *size* argument is the number of contiguous bytes to be locked or unlocked. The section to be
 23138 locked or unlocked starts at the current offset in the file and extends forward for a positive size
 23139 or backward for a negative size (the preceding bytes up to but not including the current offset).
 23140 If *size* is 0, the section from the current offset through the largest possible file offset shall be
 23141 locked (that is, from the current offset through the present or any future end-of-file). An area
 23142 need not be allocated to the file to be locked because locks may exist past the end-of-file.

23143 The sections locked with F_LOCK or F_TLOCK may, in whole or in part, contain or be contained
 23144 by a previously locked section for the same process. When this occurs, or if adjacent locked
 23145 sections would occur, the sections shall be combined into a single locked section. If the request
 23146 would cause the number of locks to exceed a system-imposed limit, the request shall fail.

23147 F_LOCK and F_TLOCK requests differ only by the action taken if the section is not available.
 23148 F_LOCK shall block the calling thread until the section is available. F_TLOCK shall cause the
 23149 function to fail if the section is already locked by another process.

23150 File locks shall be released on first close by the locking process of any file descriptor for the file.

23151 F_ULOCK requests may release (wholly or in part) one or more locked sections controlled by the
 23152 process. Locked sections shall be unlocked starting at the current file offset through *size* bytes or
 23153 to the end-of-file if *size* is (off_t)0. When all of a locked section is not released (that is, when the
 23154 beginning or end of the area to be unlocked falls within a locked section), the remaining portions
 23155 of that section shall remain locked by the process. Releasing the center portion of a locked

23156 section shall cause the remaining locked beginning and end portions to become two separate
 23157 locked sections. If the request would cause the number of locks in the system to exceed a
 23158 system-imposed limit, the request shall fail.

23159 A potential for deadlock occurs if the threads of a process controlling a locked section are
 23160 blocked by accessing another process' locked section. If the system detects that deadlock would
 23161 occur, *lockf()* shall fail with an [EDEADLK] error.

23162 The interaction between *fcntl()* and *lockf()* locks is unspecified.

23163 Blocking on a section shall be interrupted by any signal.

23164 An F_ULOCK request in which *size* is non-zero and the offset of the last byte of the requested
 23165 section is the maximum value for an object of type *off_t*, when the process has an existing lock
 23166 in which *size* is 0 and which includes the last byte of the requested section, shall be treated as a
 23167 request to unlock from the start of the requested section with a size equal to 0. Otherwise, an
 23168 F_ULOCK request shall attempt to unlock only the requested section.

23169 Attempting to lock a section of a file that is associated with a buffered stream produces
 23170 unspecified results.

23171 RETURN VALUE

23172 Upon successful completion, *lockf()* shall return 0. Otherwise, it shall return -1, set *errno* to
 23173 indicate an error, and existing locks shall not be changed.

23174 ERRORS

23175 The *lockf()* function shall fail if:

23176 [EBADF] The *fildev* argument is not a valid open file descriptor; or *function* is F_LOCK
 23177 or F_TLOCK and *fildev* is not a valid file descriptor open for writing.

23178 [EACCES] or [EAGAIN]

23179 The *function* argument is F_TLOCK or F_TEST and the section is already
 23180 locked by another process.

23181 [EDEADLK] The *function* argument is F_LOCK and a deadlock is detected.

23182 [EINTR] A signal was caught during execution of the function.

23183 [EINVAL] The *function* argument is not one of F_LOCK, F_TLOCK, F_TEST, or
 23184 F_ULOCK; or *size* plus the current file offset is less than 0.

23185 [EOVERFLOW] The offset of the first, or if *size* is not 0 then the last, byte in the requested
 23186 section cannot be represented correctly in an object of type *off_t*.

23187 The *lockf()* function may fail if:

23188 [EAGAIN] The *function* argument is F_LOCK or F_TLOCK and the file is mapped with
 23189 *mmap()*.

23190 [EDEADLK] or [ENOLCK]

23191 The *function* argument is F_LOCK, F_TLOCK, or F_ULOCK, and the request
 23192 would cause the number of locks to exceed a system-imposed limit.

23193 [EOPNOTSUPP] or [EINVAL]

23194 The implementation does not support the locking of files of the type indicated
 23195 by the *fildev* argument.

23196 **EXAMPLES**23197 **Locking a Portion of a File**

23198 In the following example, a file named `/home/cnd/mod1` is being modified. Other processes that
 23199 use locking are prevented from changing it during this process. Only the first 10 000 bytes are
 23200 locked, and the lock call fails if another process has any part of this area locked already.

```
23201 #include <fcntl.h>
23202 #include <unistd.h>

23203 int fildes;
23204 int status;
23205 ...
23206 fildes = open("/home/cnd/mod1", O_RDWR);
23207 status = lockf(fildes, F_TLOCK, (off_t)10000);
```

23208 **APPLICATION USAGE**

23209 Record-locking should not be used in combination with the *fopen()*, *fread()*, *fwrite()*, and other
 23210 *stdio* functions. Instead, the more primitive, non-buffered functions (such as *open()*) should be
 23211 used. Unexpected results may occur in processes that do buffering in the user address space. The
 23212 process may later read/write data which is/was locked. The *stdio* functions are the most
 23213 common source of unexpected buffering.

23214 The *alarm()* function may be used to provide a timeout facility in applications requiring it.

23215 **RATIONALE**

23216 None.

23217 **FUTURE DIRECTIONS**

23218 None.

23219 **SEE ALSO**

23220 *alarm()*, *chmod()*, *close()*, *creat()*, *fcntl()*, *fopen()*, *mmap()*, *open()*, *read()*, *write()*, the Base
 23221 Definitions volume of IEEE Std 1003.1-2001, **<unistd.h>**

23222 **CHANGE HISTORY**

23223 First released in Issue 4, Version 2.

23224 **Issue 5**

23225 Moved from X/OPEN UNIX extension to BASE.

23226 Large File Summit extensions are added. In particular, the description of [EINVAL] is clarified
 23227 and moved from optional to mandatory status.

23228 A note is added to the DESCRIPTION indicating the effects of attempting to lock a section of a
 23229 file that is associated with a buffered stream.

23230 **Issue 6**

23231 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

23232 **NAME**

23233 log, logf, logl — natural logarithm function

23234 **SYNOPSIS**

23235 #include <math.h>

23236 double log(double x);

23237 float logf(float x);

23238 long double logl(long double x);

23239 **DESCRIPTION**

23240 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 23241 conflict between the requirements described here and the ISO C standard is unintentional. This
 23242 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

23243 These functions shall compute the natural logarithm of their argument x , $\log_e(x)$.

23244 An application wishing to check for error situations should set *errno* to zero and call
 23245 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 23246 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 23247 zero, an error has occurred.

23248 **RETURN VALUE**23249 Upon successful completion, these functions shall return the natural logarithm of x .

23250 If x is ± 0 , a pole error shall occur and *log*(), *logf*(), and *logl*() shall return $-\text{HUGE_VAL}$,
 23251 $-\text{HUGE_VALF}$, and $-\text{HUGE_VALL}$, respectively.

23252 MX For finite values of x that are less than 0, or if x is $-\text{Inf}$, a domain error shall occur, and either a
 23253 NaN (if supported), or an implementation-defined value shall be returned.

23254 MX If x is NaN, a NaN shall be returned.23255 If x is 1, $+0$ shall be returned.23256 If x is $+\text{Inf}$, x shall be returned.23257 **ERRORS**

23258 These functions shall fail if:

23259 MX Domain Error The finite value of x is negative, or x is $-\text{Inf}$.

23260 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 23261 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling
 23262 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 23263 shall be raised.

23264 Pole Error The value of x is zero.

23265 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 23266 then *errno* shall be set to [ERANGE]. If the integer expression
 23267 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-
 23268 zero floating-point exception shall be raised.

23269 **EXAMPLES**

23270 None.

23271 **APPLICATION USAGE**

23272 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
23273 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

23274 **RATIONALE**

23275 None.

23276 **FUTURE DIRECTIONS**

23277 None.

23278 **SEE ALSO**

23279 *exp()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, *log10()*, *log1p()*, the Base Definitions volume of
23280 IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
23281 <math.h>

23282 **CHANGE HISTORY**

23283 First released in Issue 1. Derived from Issue 1 of the SVID.

23284 **Issue 5**

23285 The DESCRIPTION is updated to indicate how an application should check for an error. This
23286 text was previously published in the APPLICATION USAGE section.

23287 **Issue 6**

23288 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

23289 The *logf()* and *logl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

23290 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
23291 revised to align with the ISO/IEC 9899:1999 standard.

23292 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
23293 marked.

23294 **NAME**

23295 log10, log10f, log10l — base 10 logarithm function

23296 **SYNOPSIS**

23297 #include <math.h>

23298 double log10(double x);

23299 float log10f(float x);

23300 long double log10l(long double x);

23301 **DESCRIPTION**

23302 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 23303 conflict between the requirements described here and the ISO C standard is unintentional. This
 23304 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

23305 These functions shall compute the base 10 logarithm of their argument x , $\log_{10}(x)$.

23306 An application wishing to check for error situations should set *errno* to zero and call
 23307 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 23308 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 23309 zero, an error has occurred.

23310 **RETURN VALUE**23311 Upon successful completion, these functions shall return the base 10 logarithm of x .

23312 If x is ± 0 , a pole error shall occur and *log10()*, *log10f()*, and *log10l()* shall return $-\text{HUGE_VAL}$,
 23313 $-\text{HUGE_VALF}$, and $-\text{HUGE_VALL}$, respectively.

23314 **MX** For finite values of x that are less than 0, or if x is $-\text{Inf}$, a domain error shall occur, and either a
 23315 NaN (if supported), or an implementation-defined value shall be returned.

23316 **MX** If x is NaN, a NaN shall be returned.

23317 If x is 1, $+0$ shall be returned.

23318 If x is $+\text{Inf}$, $+\text{Inf}$ shall be returned.

23319 **ERRORS**

23320 These functions shall fail if:

23321 **MX** Domain Error The finite value of x is negative, or x is $-\text{Inf}$.

23322 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 23323 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling
 23324 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 23325 shall be raised.

23326 Pole Error The value of x is zero.

23327 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 23328 then *errno* shall be set to [ERANGE]. If the integer expression
 23329 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-
 23330 zero floating-point exception shall be raised.

23331 **EXAMPLES**

23332 None.

23333 **APPLICATION USAGE**

23334 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
23335 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

23336 **RATIONALE**

23337 None.

23338 **FUTURE DIRECTIONS**

23339 None.

23340 **SEE ALSO**

23341 *feclearexcept()*, *fetestexcept()*, *isnan()*, *log()*, *pow()*, the Base Definitions volume of
23342 IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
23343 <math.h>

23344 **CHANGE HISTORY**

23345 First released in Issue 1. Derived from Issue 1 of the SVID.

23346 **Issue 5**

23347 The DESCRIPTION is updated to indicate how an application should check for an error. This
23348 text was previously published in the APPLICATION USAGE section.

23349 **Issue 6**

23350 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

23351 The *log10f()* and *log10l()* functions are added for alignment with the ISO/IEC 9899:1999
23352 standard.

23353 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
23354 revised to align with the ISO/IEC 9899:1999 standard.

23355 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
23356 marked.

23357 **NAME**

23358 log1p, log1pf, log1pl — compute a natural logarithm

23359 **SYNOPSIS**

23360 #include <math.h>

23361 double log1p(double x);

23362 float log1pf(float x);

23363 long double log1pl(long double x);

23364 **DESCRIPTION**

23365 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 23366 conflict between the requirements described here and the ISO C standard is unintentional. This
 23367 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

23368 These functions shall compute $\log_e(1.0 + x)$.

23369 An application wishing to check for error situations should set *errno* to zero and call
 23370 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 23371 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 23372 zero, an error has occurred.

23373 **RETURN VALUE**23374 Upon successful completion, these functions shall return the natural logarithm of $1.0 + x$.

23375 If x is -1 , a pole error shall occur and *log1p()*, *log1pf()*, and *log1pl()* shall return $-\text{HUGE_VAL}$,
 23376 $-\text{HUGE_VALF}$, and $-\text{HUGE_VALL}$, respectively.

23377 **MX** For finite values of x that are less than -1 , or if x is $-\text{Inf}$, a domain error shall occur, and either a
 23378 NaN (if supported), or an implementation-defined value shall be returned.

23379 **MX** If x is NaN, a NaN shall be returned.23380 If x is ± 0 , or $+\text{Inf}$, x shall be returned.23381 If x is subnormal, a range error may occur and x should be returned.23382 **ERRORS**

23383 These functions shall fail if:

23384 **MX** Domain Error The finite value of x is less than -1 , or x is $-\text{Inf}$.

23385 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 23386 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling
 23387 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 23388 shall be raised.

23389 Pole Error The value of x is -1 .

23390 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 23391 then *errno* shall be set to [ERANGE]. If the integer expression
 23392 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-
 23393 zero floating-point exception shall be raised.

23394 These functions may fail if:

23395 **MX** Range Error The value of x is subnormal.

23396 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 23397 then *errno* shall be set to [ERANGE]. If the integer expression
 23398 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
 23399 floating-point exception shall be raised.

23400 **EXAMPLES**

23401 None.

23402 **APPLICATION USAGE**

23403 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
23404 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

23405 **RATIONALE**

23406 None.

23407 **FUTURE DIRECTIONS**

23408 None.

23409 **SEE ALSO**

23410 *feclearexcept()*, *fetestexcept()*, *log()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section
23411 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

23412 **CHANGE HISTORY**

23413 First released in Issue 4, Version 2.

23414 **Issue 5**

23415 Moved from X/OPEN UNIX extension to BASE.

23416 **Issue 6**

23417 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

23418 The *log1p()* function is no longer marked as an extension.

23419 The *log1pf()* and *log1pl()* functions are added for alignment with the ISO/IEC 9899:1999
23420 standard.

23421 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
23422 revised to align with the ISO/IEC 9899:1999 standard.

23423 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
23424 marked.

23425 **NAME**

23426 log2, log2f, log2l — compute base 2 logarithm functions

23427 **SYNOPSIS**

23428 #include <math.h>

23429 double log2(double x);

23430 float log2f(float x);

23431 long double log2l(long double x);

23432 **DESCRIPTION**

23433 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 23434 conflict between the requirements described here and the ISO C standard is unintentional. This
 23435 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

23436 These functions shall compute the base 2 logarithm of their argument x , $\log_2(x)$.

23437 An application wishing to check for error situations should set *errno* to zero and call
 23438 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 23439 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 23440 zero, an error has occurred.

23441 **RETURN VALUE**23442 Upon successful completion, these functions shall return the base 2 logarithm of x .

23443 If x is ± 0 , a pole error shall occur and *log2()*, *log2f()*, and *log2l()* shall return $-\text{HUGE_VAL}$,
 23444 $-\text{HUGE_VALF}$, and $-\text{HUGE_VALL}$, respectively.

23445 **MX** For finite values of x that are less than 0, or if x is $-\text{Inf}$, a domain error shall occur, and either a
 23446 NaN (if supported), or an implementation-defined value shall be returned.

23447 **MX** If x is NaN, a NaN shall be returned.23448 If x is 1, $+0$ shall be returned.23449 If x is $+\text{Inf}$, x shall be returned.23450 **ERRORS**

23451 These functions shall fail if:

23452 **MX** Domain Error The finite value of x is less than zero, or x is $-\text{Inf}$.

23453 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 23454 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling
 23455 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 23456 shall be raised.

23457 Pole Error The value of x is zero.

23458 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 23459 then *errno* shall be set to [ERANGE]. If the integer expression
 23460 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-
 23461 zero floating-point exception shall be raised.

23462 EXAMPLES

23463 None.

23464 APPLICATION USAGE

23465 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
23466 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

23467 RATIONALE

23468 None.

23469 FUTURE DIRECTIONS

23470 None.

23471 SEE ALSO

23472 *feclearexcept()*, *fetetestexcept()*, *log()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section
23473 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

23474 CHANGE HISTORY

23475 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

23476 **NAME**

23477 logb, logbf, logbl — radix-independent exponent

23478 **SYNOPSIS**

23479 #include <math.h>

23480 double logb(double x);

23481 float logbf(float x);

23482 long double logbl(long double x);

23483 **DESCRIPTION**

23484 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 23485 conflict between the requirements described here and the ISO C standard is unintentional. This
 23486 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

23487 These functions shall compute the exponent of x , which is the integral part of $\log_r |x|$, as a
 23488 signed floating-point value, for non-zero x , where r is the radix of the machine's floating-point
 23489 arithmetic, which is the value of FLT_RADIX defined in the <float.h> header.

23490 If x is subnormal it is treated as though it were normalized; thus for finite positive x :

23491 $1 \leq x * \text{FLT_RADIX}^{-\text{logb}(x)} < \text{FLT_RADIX}$

23492 An application wishing to check for error situations should set *errno* to zero and call
 23493 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 23494 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 23495 zero, an error has occurred.

23496 **RETURN VALUE**

23497 Upon successful completion, these functions shall return the exponent of x .

23498 If x is ± 0 , a pole error shall occur and *logb*(), *logbf*(), and *logbl*() shall return -HUGE_VAL,
 23499 -HUGE_VALF, and -HUGE_VALL, respectively.

23500 **MX** If x is NaN, a NaN shall be returned.

23501 If x is $\pm\text{Inf}$, $+\text{Inf}$ shall be returned.

23502 **ERRORS**

23503 These functions shall fail if:

23504 Pole Error The value of x is ± 0 .

23505 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 23506 then *errno* shall be set to [ERANGE]. If the integer expression
 23507 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-
 23508 zero floating-point exception shall be raised.

23509 **EXAMPLES**

23510 None.

23511 **APPLICATION USAGE**

23512 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
 23513 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

23514 **RATIONALE**

23515 None.

23516 **FUTURE DIRECTIONS**

23517 None.

23518 **SEE ALSO**

23519 *feclearexcept()*, *fetestexcept()*, *ilogb()*, *scalb()*, the Base Definitions volume of IEEE Std 1003.1-2001,
23520 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <float.h>, <math.h>

23521 **CHANGE HISTORY**

23522 First released in Issue 4, Version 2.

23523 **Issue 5**

23524 Moved from X/OPEN UNIX extension to BASE.

23525 **Issue 6**23526 The *logb()* function is no longer marked as an extension.23527 The *logbf()* and *logbl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

23528 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
23529 revised to align with the ISO/IEC 9899:1999 standard.

23530 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
23531 marked.

23532 **NAME**

23533 logf, logl — natural logarithm function

23534 **SYNOPSIS**

23535 #include <math.h>

23536 float logf(float x);

23537 long double logl(long double x);

23538 **DESCRIPTION**23539 Refer to *log()*.

23540 NAME

23541 longjmp — non-local goto

23542 SYNOPSIS

23543 #include <setjmp.h>

23544 void longjmp(jmp_buf env, int val);

23545 DESCRIPTION

23546 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 23547 conflict between the requirements described here and the ISO C standard is unintentional. This
 23548 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

23549 The *longjmp()* function shall restore the environment saved by the most recent invocation of
 23550 *setjmp()* in the same thread, with the corresponding **jmp_buf** argument. If there is no such
 23551 invocation, or if the function containing the invocation of *setjmp()* has terminated execution in
 23552 the interim, or if the invocation of *setjmp()* was within the scope of an identifier with variably
 23553 CX modified type and execution has left that scope in the interim, the behavior is undefined. It is
 23554 unspecified whether *longjmp()* restores the signal mask, leaves the signal mask unchanged, or
 23555 restores it to its value at the time *setjmp()* was called.

23556 All accessible objects have values, and all other components of the abstract machine have state
 23557 (for example, floating-point status flags and open files), as of the time *longjmp()* was called,
 23558 except that the values of objects of automatic storage duration are unspecified if they meet all
 23559 the following conditions:

- 23560 • They are local to the function containing the corresponding *setjmp()* invocation.
- 23561 • They do not have volatile-qualified type.
- 23562 • They are changed between the *setjmp()* invocation and *longjmp()* call.

23563 CX As it bypasses the usual function call and return mechanisms, *longjmp()* shall execute correctly
 23564 in contexts of interrupts, signals, and any of their associated functions. However, if *longjmp()* is
 23565 invoked from a nested signal handler (that is, from a function invoked as a result of a signal
 23566 raised during the handling of another signal), the behavior is undefined.

23567 The effect of a call to *longjmp()* where initialization of the **jmp_buf** structure was not performed
 23568 in the calling thread is undefined.

23569 RETURN VALUE

23570 After *longjmp()* is completed, program execution continues as if the corresponding invocation of
 23571 *setjmp()* had just returned the value specified by *val*. The *longjmp()* function shall not cause
 23572 *setjmp()* to return 0; if *val* is 0, *setjmp()* shall return 1.

23573 ERRORS

23574 No errors are defined.

23575 EXAMPLES

23576 None.

23577 APPLICATION USAGE

23578 Applications whose behavior depends on the value of the signal mask should not use *longjmp()*
 23579 and *setjmp()*, since their effect on the signal mask is unspecified, but should instead use the
 23580 *siglongjmp()* and *sigsetjmp()* functions (which can save and restore the signal mask under
 23581 application control).

23582 **RATIONALE**

23583 None.

23584 **FUTURE DIRECTIONS**

23585 None.

23586 **SEE ALSO**

23587 *setjmp()*, *sigaction()*, *siglongjmp()*, *sigsetjmp()*, the Base Definitions volume of
23588 IEEE Std 1003.1-2001, <**setjmp.h**>

23589 **CHANGE HISTORY**

23590 First released in Issue 1. Derived from Issue 1 of the SVID.

23591 **Issue 5**

23592 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

23593 **Issue 6**

23594 Extensions beyond the ISO C standard are marked.

23595 The following new requirements on POSIX implementations derive from alignment with the
23596 Single UNIX Specification:

- 23597 • The DESCRIPTION now explicitly makes *longjmp()*'s effect on the signal mask unspecified.

23598 The DESCRIPTION is updated for alignment with the ISO/IEC 9899:1999 standard.

23599 NAME

23600 lrand48 — generate uniformly distributed pseudo-random non-negative long integers

23601 SYNOPSIS

23602 xSI #include <stdlib.h>

23603 long lrand48(void);

23604

23605 DESCRIPTION

23606 Refer to *drand48()*.

23607 **NAME**

23608 lrint, lrintf, lrintl — round to nearest integer value using current rounding direction

23609 **SYNOPSIS**

23610 #include <math.h>

23611 long lrint(double x);

23612 long lrintf(float x);

23613 long lrintl(long double x);

23614 **DESCRIPTION**

23615 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 23616 conflict between the requirements described here and the ISO C standard is unintentional. This
 23617 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

23618 These functions shall round their argument to the nearest integer value, rounding according to
 23619 the current rounding direction.

23620 An application wishing to check for error situations should set *errno* to zero and call
 23621 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 23622 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 23623 zero, an error has occurred.

23624 **RETURN VALUE**

23625 Upon successful completion, these functions shall return the rounded integer value.

23626 **MX** If *x* is NaN, a domain error shall occur and an unspecified value is returned.23627 If *x* is +Inf, a domain error shall occur and an unspecified value is returned.23628 If *x* is -Inf, a domain error shall occur and an unspecified value is returned.

23629 If the correct value is positive and too large to represent as a **long**, a domain error shall occur
 23630 and an unspecified value is returned.

23631 If the correct value is negative and too large to represent as a **long**, a domain error shall occur
 23632 and an unspecified value is returned.

23633 **ERRORS**

23634 These functions shall fail if:

23635 MX	Domain Error	The <i>x</i> argument is NaN or \pm Inf, or the correct value is not representable as an integer.
-----------------	---------------------	---

23637	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
23638	then <i>errno</i> shall be set to [EDOM]. If the integer expression (math_errhandling
23639	& MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
23640	shall be raised.

23641 **EXAMPLES**

23642 None.

23643 **APPLICATION USAGE**

23644 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
 23645 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

23646 **RATIONALE**

23647 These functions provide floating-to-integer conversions. They round according to the current
 23648 rounding direction. If the rounded value is outside the range of the return type, the numeric
 23649 result is unspecified and the invalid floating-point exception is raised. When they raise no other
 23650 floating-point exception and the result differs from the argument, they raise the inexact

23651 floating-point exception.

23652 **FUTURE DIRECTIONS**

23653 None.

23654 **SEE ALSO**

23655 *feclearexcept()*, *fetestexcept()*, *llrint()*, the Base Definitions volume of IEEE Std 1003.1-2001,
23656 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <**math.h**>

23657 **CHANGE HISTORY**

23658 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

23659 **NAME**

23660 lround, lroundf, lroundl — round to nearest integer value

23661 **SYNOPSIS**

23662 #include <math.h>

23663 long lround(double x);

23664 long lroundf(float x);

23665 long lroundl(long double x);

23666 **DESCRIPTION**

23667 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 23668 conflict between the requirements described here and the ISO C standard is unintentional. This
 23669 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

23670 These functions shall round their argument to the nearest integer value, rounding halfway cases
 23671 away from zero, regardless of the current rounding direction.

23672 An application wishing to check for error situations should set *errno* to zero and call
 23673 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 23674 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 23675 zero, an error has occurred.

23676 **RETURN VALUE**

23677 Upon successful completion, these functions shall return the rounded integer value.

23678 **MX** If *x* is NaN, a domain error shall occur and an unspecified value is returned.23679 If *x* is +Inf, a domain error shall occur and an unspecified value is returned.23680 If *x* is -Inf, a domain error shall occur and an unspecified value is returned.

23681 If the correct value is positive and too large to represent as a **long**, a domain error shall occur
 23682 and an unspecified value is returned.

23683 If the correct value is negative and too large to represent as a **long**, a domain error shall occur
 23684 and an unspecified value is returned.

23685 **ERRORS**

23686 These functions shall fail if:

23687 MX	Domain Error	The <i>x</i> argument is NaN or \pm Inf, or the correct value is not representable as an integer.
-----------------	---------------------	---

23689	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
23690	then <i>errno</i> shall be set to [EDOM]. If the integer expression (math_errhandling
23691	& MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
23692	shall be raised.

23693 **EXAMPLES**

23694 None.

23695 **APPLICATION USAGE**

23696 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
 23697 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

23698 **RATIONALE**

23699 These functions differ from the *lrint*() functions in the default rounding direction, with the
 23700 *lround*() functions rounding halfway cases away from zero and needing not to raise the inexact
 23701 floating-point exception for non-integer arguments that round to within the range of the return
 23702 type.

23703 FUTURE DIRECTIONS

23704 None.

23705 SEE ALSO

23706 *feclearexcept()*, *fetestexcept()*, *llround()*, the Base Definitions volume of IEEE Std 1003.1-2001,
23707 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

23708 CHANGE HISTORY

23709 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

23710 **NAME**

23711 lsearch, lfind — linear search and update

23712 **SYNOPSIS**

23713 XSI #include <search.h>

```

23714 void *lsearch(const void *key, void *base, size_t *nel, size_t width,
23715             int (*compar)(const void *, const void *));
23716 void *lfind(const void *key, const void *base, size_t *nel,
23717            size_t width, int (*compar)(const void *, const void *));
23718 
```

23719 **DESCRIPTION**

23720 The *lsearch()* function shall linearly search the table and return a pointer into the table for the
 23721 matching entry. If the entry does not occur, it shall be added at the end of the table. The *key*
 23722 argument points to the entry to be sought in the table. The *base* argument points to the first
 23723 element in the table. The *width* argument is the size of an element in bytes. The *nel* argument
 23724 points to an integer containing the current number of elements in the table. The integer to which
 23725 *nel* points shall be incremented if the entry is added to the table. The *compar* argument points to
 23726 a comparison function which the application shall supply (for example, *strcmp()*). It is called
 23727 with two arguments that point to the elements being compared. The application shall ensure
 23728 that the function returns 0 if the elements are equal, and non-zero otherwise.

23729 The *lfind()* function shall be equivalent to *lsearch()*, except that if the entry is not found, it is not
 23730 added to the table. Instead, a null pointer is returned.

23731 **RETURN VALUE**

23732 If the searched for entry is found, both *lsearch()* and *lfind()* shall return a pointer to it. Otherwise,
 23733 *lfind()* shall return a null pointer and *lsearch()* shall return a pointer to the newly added element.

23734 Both functions shall return a null pointer in case of error.

23735 **ERRORS**

23736 No errors are defined.

23737 **EXAMPLES**23738 **Storing Strings in a Table**

23739 This fragment reads in less than or equal to TABSIZE strings of length less than or equal to
 23740 ELSIZE and stores them in a table, eliminating duplicates.

```

23741 #include <stdio.h>
23742 #include <string.h>
23743 #include <search.h>
23744 #define TABSIZE 50
23745 #define ELSIZE 120
23746 ...
23747     char line[ELSIZE], tab[TABSIZE][ELSIZE];
23748     size_t nel = 0;
23749     ...
23750     while (fgets(line, ELSIZE, stdin) != NULL && nel < TABSIZE)
23751         (void) lsearch(line, tab, &nel,
23752                       ELSIZE, (int (*)(const void *, const void *)) strcmp);
23753     ...

```


23754 Finding a Matching Entry

23755 The following example finds any line that reads "This is a test.".

```
23756 #include <search.h>
23757 #include <string.h>
23758 ...
23759 char line[ELSIZE], tab[TABSIZE][ELSIZE];
23760 size_t nel = 0;
23761 char *findline;
23762 void *entry;

23763 findline = "This is a test.\n";

23764 entry = lfind(findline, tab, &nel, ELSIZE, (
23765     int (*)(const void *, const void *)) strcmp);
```

23766 APPLICATION USAGE

23767 The comparison function need not compare every byte, so arbitrary data may be contained in
23768 the elements in addition to the values being compared.

23769 Undefined results can occur if there is not enough room in the table to add a new item.

23770 RATIONALE

23771 None.

23772 FUTURE DIRECTIONS

23773 None.

23774 SEE ALSO

23775 *hcreate()*, *tsearch()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**search.h**>

23776 CHANGE HISTORY

23777 First released in Issue 1. Derived from Issue 1 of the SVID.

23778 Issue 6

23779 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

23780 **NAME**

23781 lseek — move the read/write file offset

23782 **SYNOPSIS**

23783 #include <unistd.h>

23784 off_t lseek(int *fildes*, off_t *offset*, int *whence*);23785 **DESCRIPTION**

23786 The *lseek()* function shall set the file offset for the open file description associated with the file
 23787 descriptor *fildes*, as follows:

- 23788 • If *whence* is SEEK_SET, the file offset shall be set to *offset* bytes.
- 23789 • If *whence* is SEEK_CUR, the file offset shall be set to its current location plus *offset*.
- 23790 • If *whence* is SEEK_END, the file offset shall be set to the size of the file plus *offset*.

23791 The symbolic constants SEEK_SET, SEEK_CUR, and SEEK_END are defined in <unistd.h>.

23792 The behavior of *lseek()* on devices which are incapable of seeking is implementation-defined.
 23793 The value of the file offset associated with such a device is undefined.

23794 The *lseek()* function shall allow the file offset to be set beyond the end of the existing data in the
 23795 file. If data is later written at this point, subsequent reads of data in the gap shall return bytes
 23796 with the value 0 until data is actually written into the gap.

23797 The *lseek()* function shall not, by itself, extend the size of a file.23798 SHM If *fildes* refers to a shared memory object, the result of the *lseek()* function is unspecified.23799 TYM If *fildes* refers to a typed memory object, the result of the *lseek()* function is unspecified.23800 **RETURN VALUE**

23801 Upon successful completion, the resulting offset, as measured in bytes from the beginning of the
 23802 file, shall be returned. Otherwise, (off_t)−1 shall be returned, *errno* shall be set to indicate the
 23803 error, and the file offset shall remain unchanged.

23804 **ERRORS**23805 The *lseek()* function shall fail if:

- | | | |
|-------|-------------|--|
| 23806 | [EBADF] | The <i>fildes</i> argument is not an open file descriptor. |
| 23807 | [EINVAL] | The <i>whence</i> argument is not a proper value, or the resulting file offset would |
| 23808 | | be negative for a regular file, block special file, or directory. |
| 23809 | [EOVERFLOW] | The resulting file offset would be a value which cannot be represented |
| 23810 | | correctly in an object of type off_t . |
| 23811 | [ESPIPE] | The <i>fildes</i> argument is associated with a pipe, FIFO, or socket. |

23812 **EXAMPLES**

23813 None.

23814 **APPLICATION USAGE**

23815 None.

23816 **RATIONALE**

23817 The ISO C standard includes the functions *fgetpos()* and *fsetpos()*, which work on very large files
 23818 by use of a special positioning type.

23819 Although *lseek()* may position the file offset beyond the end of the file, this function does not
 23820 itself extend the size of the file. While the only function in IEEE Std 1003.1-2001 that may directly

23821 extend the size of the file is *write()*, *truncate()*, and *ftruncate()*, several functions originally
 23822 derived from the ISO C standard, such as *fwrite()*, *fprintf()*, and so on, may do so (by causing
 23823 calls on *write()*).

23824 An invalid file offset that would cause [EINVAL] to be returned may be both implementation-
 23825 defined and device-dependent (for example, memory may have few invalid values). A negative
 23826 file offset may be valid for some devices in some implementations.

23827 The POSIX.1-1990 standard did not specifically prohibit *lseek()* from returning a negative offset.
 23828 Therefore, an application was required to clear *errno* prior to the call and check *errno* upon return
 23829 to determine whether a return value of (*off_t*)−1 is a negative offset or an indication of an error
 23830 condition. The standard developers did not wish to require this action on the part of a
 23831 conforming application, and chose to require that *errno* be set to [EINVAL] when the resulting
 23832 file offset would be negative for a regular file, block special file, or directory.

23833 FUTURE DIRECTIONS

23834 None.

23835 SEE ALSO

23836 *open()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**sys/types.h**>, <**unistd.h**>

23837 CHANGE HISTORY

23838 First released in Issue 1. Derived from Issue 1 of the SVID.

23839 Issue 5

23840 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

23841 Large File Summit extensions are added.

23842 Issue 6

23843 In the SYNOPSIS, the optional include of the <**sys/types.h**> header is removed.

23844 The following new requirements on POSIX implementations derive from alignment with the
 23845 Single UNIX Specification:

- 23846 • The requirement to include <**sys/types.h**> has been removed. Although <**sys/types.h**> was
 23847 required for conforming implementations of previous POSIX specifications, it was not
 23848 required for UNIX applications.
- 23849 • The [EOVERFLOW] error condition is added. This change is to support large files.

23850 An additional [ESPIPE] error condition is added for sockets.

23851 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that
 23852 *lseek()* results are unspecified for typed memory objects.

23853 **NAME**23854 **lstat** — get symbolic link status23855 **SYNOPSIS**23856 `#include <sys/stat.h>`23857 `int lstat(const char *restrict path, struct stat *restrict buf);`23858 **DESCRIPTION**

23859 The *lstat()* function shall be equivalent to *stat()*, except when *path* refers to a symbolic link. In
 23860 that case *lstat()* shall return information about the link, while *stat()* shall return information
 23861 about the file the link references.

23862 For symbolic links, the *st_mode* member shall contain meaningful information when used with
 23863 the file type macros, and the *st_size* member shall contain the length of the pathname contained
 23864 in the symbolic link. File mode bits and the contents of the remaining members of the **stat**
 23865 structure are unspecified. The value returned in the *st_size* member is the length of the contents
 23866 of the symbolic link, and does not count any trailing null.

23867 **RETURN VALUE**

23868 Upon successful completion, *lstat()* shall return 0. Otherwise, it shall return -1 and set *errno* to
 23869 indicate the error.

23870 **ERRORS**23871 The *lstat()* function shall fail if:

23872 [EACCES] A component of the path prefix denies search permission.

23873 [EIO] An error occurred while reading from the file system.

23874 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 23875 argument.

23876 [ENAMETOOLONG]

23877 The length of a pathname exceeds {PATH_MAX} or a pathname component is
 23878 longer than {NAME_MAX}.

23879 [ENOTDIR] A component of the path prefix is not a directory.

23880 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

23881 [EOVERFLOW] The file size in bytes or the number of blocks allocated to the file or the file
 23882 serial number cannot be represented correctly in the structure pointed to by
 23883 *buf*.

23884 The *lstat()* function may fail if:

23885 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 23886 resolution of the *path* argument.

23887 [ENAMETOOLONG]

23888 As a result of encountering a symbolic link in resolution of the *path* argument,
 23889 the length of the substituted pathname string exceeded {PATH_MAX}.

23890 [EOVERFLOW] One of the members is too large to store into the structure pointed to by the
 23891 *buf* argument.

23892 **EXAMPLES**23893 **Obtaining Symbolic Link Status Information**

23894 The following example shows how to obtain status information for a symbolic link named
23895 **/modules/pass1**. The structure variable *buffer* is defined for the **stat** structure. If the *path*
23896 argument specified the filename for the file pointed to by the symbolic link (**/home/cnd/mod1**),
23897 the results of calling the function would be the same as those returned by a call to the *stat()*
23898 function.

```
23899 #include <sys/stat.h>
23900 struct stat buffer;
23901 int status;
23902 ...
23903 status = lstat("/modules/pass1", &buffer);
```

23904 **APPLICATION USAGE**

23905 None.

23906 **RATIONALE**

23907 The *lstat()* function is not required to update the time-related fields if the named file is not a
23908 symbolic link. While the *st_uid*, *st_gid*, *st_atime*, *st_mtime*, and *st_ctime* members of the **stat**
23909 structure may apply to a symbolic link, they are not required to do so. No functions in
23910 IEEE Std 1003.1-2001 are required to maintain any of these time fields.

23911 **FUTURE DIRECTIONS**

23912 None.

23913 **SEE ALSO**

23914 *lstat()*, *readlink()*, *stat()*, *symlink()*, the Base Definitions volume of IEEE Std 1003.1-2001,
23915 **<sys/stat.h>**

23916 **CHANGE HISTORY**

23917 First released in Issue 4, Version 2.

23918 **Issue 5**

23919 Moved from X/OPEN UNIX extension to BASE.

23920 Large File Summit extensions are added.

23921 **Issue 6**

23922 The following changes were made to align with the IEEE P1003.1a draft standard:

- 23923 • This function is now mandatory.
- 23924 • The [ELOOP] optional error condition is added.

23925 The **restrict** keyword is added to the *lstat()* prototype for alignment with the ISO/IEC 9899:1999
23926 standard.

23927 **NAME**

23928 makecontext, swapcontext — manipulate user contexts

23929 **SYNOPSIS**

```

23930 xSI      #include <ucontext.h>

23931      void makecontext(ucontext_t *ucp, void (*func)(void),
23932                      int argc, ...);
23933      int swapcontext(ucontext_t *restrict oucp,
23934                     const ucontext_t *restrict ucp);
23935 
```

23936 **DESCRIPTION**

23937 The *makecontext()* function shall modify the context specified by *ucp*, which has been initialized
 23938 using *getcontext()*. When this context is resumed using *swapcontext()* or *setcontext()*, program
 23939 execution shall continue by calling *func*, passing it the arguments that follow *argc* in the
 23940 *makecontext()* call.

23941 Before a call is made to *makecontext()*, the application shall ensure that the context being
 23942 modified has a stack allocated for it. The application shall ensure that the value of *argc* matches
 23943 the number of integer arguments passed to *func*; otherwise, the behavior is undefined.

23944 The *uc_link* member is used to determine the context that shall be resumed when the context
 23945 being modified by *makecontext()* returns. The application shall ensure that the *uc_link* member is
 23946 initialized prior to the call to *makecontext()*.

23947 The *swapcontext()* function shall save the current context in the context structure pointed to by
 23948 *oucp* and shall set the context to the context structure pointed to by *ucp*.

23949 **RETURN VALUE**

23950 Upon successful completion, *swapcontext()* shall return 0. Otherwise, *-1* shall be returned and
 23951 *errno* set to indicate the error.

23952 **ERRORS**

23953 The *swapcontext()* function shall fail if:

23954 [ENOMEM] The *ucp* argument does not have enough stack left to complete the operation.

23955 **EXAMPLES**

23956 The following example illustrates the use of *makecontext()*:

```

23957      #include <stdio.h>
23958      #include <ucontext.h>

23959      static ucontext_t ctx[3];

23960      static void
23961      f1 (void)
23962      {
23963          puts("start f1");
23964          swapcontext(&ctx[1], &ctx[2]);
23965          puts("finish f1");
23966      }

23967      static void
23968      f2 (void)
23969      {
23970          puts("start f2");
23971          swapcontext(&ctx[2], &ctx[1]);

```



```

23972         puts("finish f2");
23973     }
23974     int
23975     main (void)
23976     {
23977         char st1[8192];
23978         char st2[8192];
23979
23980         getcontext(&ctx[1]);
23981         ctx[1].uc_stack.ss_sp = st1;
23982         ctx[1].uc_stack.ss_size = sizeof st1;
23983         ctx[1].uc_link = &ctx[0];
23984         makecontext(&ctx[1], f1, 0);
23985
23986         getcontext(&ctx[2]);
23987         ctx[2].uc_stack.ss_sp = st2;
23988         ctx[2].uc_stack.ss_size = sizeof st2;
23989         ctx[2].uc_link = &ctx[1];
23990         makecontext(&ctx[2], f2, 0);
23991
23992         swapcontext(&ctx[0], &ctx[2]);
23993         return 0;
23994     }

```

23992 APPLICATION USAGE

23993 None.

23994 RATIONALE

23995 None.

23996 FUTURE DIRECTIONS

23997 None.

23998 SEE ALSO

23999 *exit()*, *getcontext()*, *sigaction()*, *sigprocmask()*, the Base Definitions volume of
24000 IEEE Std 1003.1-2001, <**ucontext.h**>

24001 CHANGE HISTORY

24002 First released in Issue 4, Version 2.

24003 Issue 5

24004 Moved from X/OPEN UNIX extension to BASE.

24005 In the ERRORS section, the description of [ENOMEM] is changed to apply to *swapcontext()* only.

24006 Issue 6

24007 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

24008 The **restrict** keyword is added to the *swapcontext()* prototype for alignment with the
24009 ISO/IEC 9899:1999 standard.

24010 **NAME**

24011 malloc — a memory allocator

24012 **SYNOPSIS**

24013 #include <stdlib.h>

24014 void *malloc(size_t size);

24015 **DESCRIPTION**

24016 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 24017 conflict between the requirements described here and the ISO C standard is unintentional. This
 24018 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

24019 The *malloc()* function shall allocate unused space for an object whose size in bytes is specified by
 24020 *size* and whose value is unspecified.

24021 The order and contiguity of storage allocated by successive calls to *malloc()* is unspecified. The
 24022 pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to
 24023 a pointer to any type of object and then used to access such an object in the space allocated (until
 24024 the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object
 24025 disjoint from any other object. The pointer returned points to the start (lowest byte address) of
 24026 the allocated space. If the space cannot be allocated, a null pointer shall be returned. If the size of
 24027 the space requested is 0, the behavior is implementation-defined: the value returned shall be
 24028 either a null pointer or a unique pointer.

24029 **RETURN VALUE**

24030 Upon successful completion with *size* not equal to 0, *malloc()* shall return a pointer to the
 24031 allocated space. If *size* is 0, either a null pointer or a unique pointer that can be successfully
 24032 CX passed to *free()* shall be returned. Otherwise, it shall return a null pointer and set *errno* to
 24033 indicate the error.

24034 **ERRORS**24035 The *malloc()* function shall fail if:

24036 CX [ENOMEM] Insufficient storage space is available.

24037 **EXAMPLES**

24038 None.

24039 **APPLICATION USAGE**

24040 None.

24041 **RATIONALE**

24042 None.

24043 **FUTURE DIRECTIONS**

24044 None.

24045 **SEE ALSO**24046 *calloc()*, *free()*, *realloc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>24047 **CHANGE HISTORY**

24048 First released in Issue 1. Derived from Issue 1 of the SVID.

24049 **Issue 6**

24050 Extensions beyond the ISO C standard are marked.

24051 The following new requirements on POSIX implementations derive from alignment with the
 24052 Single UNIX Specification:

- 24053 • In the RETURN VALUE section, the requirement to set *errno* to indicate an error is added.
- 24054 • The [ENOMEM] error condition is added.

24055 **NAME**

24056 mblen — get number of bytes in a character

24057 **SYNOPSIS**

24058 #include <stdlib.h>

24059 int mblen(const char *s, size_t n);

24060 **DESCRIPTION**

24061 cx The functionality described on this reference page is aligned with the ISO C standard. Any
 24062 conflict between the requirements described here and the ISO C standard is unintentional. This
 24063 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

24064 If *s* is not a null pointer, *mblen()* shall determine the number of bytes constituting the character
 24065 pointed to by *s*. Except that the shift state of *mbtowc()* is not affected, it shall be equivalent to:

24066 mbtowc((wchar_t *)0, s, n);

24067 The implementation shall behave as if no function defined in this volume of
 24068 IEEE Std 1003.1-2001 calls *mblen()*.

24069 The behavior of this function is affected by the *LC_CTYPE* category of the current locale. For a
 24070 state-dependent encoding, this function shall be placed into its initial state by a call for which its
 24071 character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null
 24072 pointer shall cause the internal state of the function to be altered as necessary. A call with *s* as a
 24073 null pointer shall cause this function to return a non-zero value if encodings have state
 24074 dependency, and 0 otherwise. If the implementation employs special bytes to change the shift
 24075 state, these bytes shall not produce separate wide-character codes, but shall be grouped with an
 24076 adjacent character. Changing the *LC_CTYPE* category causes the shift state of this function to be
 24077 unspecified.

24078 **RETURN VALUE**

24079 If *s* is a null pointer, *mblen()* shall return a non-zero or 0 value, if character encodings,
 24080 respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, *mblen()* shall
 24081 either return 0 (if *s* points to the null byte), or return the number of bytes that constitute the
 24082 character (if the next *n* or fewer bytes form a valid character), or return -1 (if they do not form a
 24083 cx valid character) and may set *errno* to indicate the error. In no case shall the value returned be
 24084 greater than *n* or the value of the {MB_CUR_MAX} macro.

24085 **ERRORS**24086 The *mblen()* function may fail if:

24087 xsi [EILSEQ] Invalid character sequence is detected.

24088 **EXAMPLES**

24089 None.

24090 **APPLICATION USAGE**

24091 None.

24092 **RATIONALE**

24093 None.

24094 **FUTURE DIRECTIONS**

24095 None.

24096 **SEE ALSO**

24097 *mbtowc()*, *mbstowcs()*, *wctomb()*, *wcstombs()*, the Base Definitions volume of
24098 IEEE Std 1003.1-2001, <stdlib.h>

24099 **CHANGE HISTORY**

24100 First released in Issue 4. Aligned with the ISO C standard.

24101 **NAME**

24102 mbrlen — get number of bytes in a character (restartable)

24103 **SYNOPSIS**

24104 #include <wchar.h>

24105 size_t mbrlen(const char *restrict *s*, size_t *n*,24106 mbstate_t *restrict *ps*);24107 **DESCRIPTION**

24108 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 24109 conflict between the requirements described here and the ISO C standard is unintentional. This
 24110 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

24111 If *s* is not a null pointer, *mbrlen()* shall determine the number of bytes constituting the character
 24112 pointed to by *s*. It shall be equivalent to:

24113 mbstate_t internal;

24114 mbrtowc(NULL, *s*, *n*, *ps* != NULL ? *ps* : &internal);

24115 If *ps* is a null pointer, the *mbrlen()* function shall use its own internal **mbstate_t** object, which is
 24116 initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t** object
 24117 pointed to by *ps* shall be used to completely describe the current conversion state of the
 24118 associated character sequence. The implementation shall behave as if no function defined in this
 24119 volume of IEEE Std 1003.1-2001 calls *mbrlen()*.

24120 The behavior of this function is affected by the *LC_CTYPE* category of the current locale.24121 **RETURN VALUE**24122 The *mbrlen()* function shall return the first of the following that applies:

24123 0 If the next *n* or fewer bytes complete the character that corresponds to the null
 24124 wide character.

24125 *positive* If the next *n* or fewer bytes complete a valid character; the value returned shall
 24126 be the number of bytes that complete the character.

24127 (**size_t**)−2 If the next *n* bytes contribute to an incomplete but potentially valid character,
 24128 and all *n* bytes have been processed. When *n* has at least the value of the
 24129 {MB_CUR_MAX} macro, this case can only occur if *s* points at a sequence of
 24130 redundant shift sequences (for implementations with state-dependent
 24131 encodings).

24132 (**size_t**)−1 If an encoding error occurs, in which case the next *n* or fewer bytes do not
 24133 contribute to a complete and valid character. In this case, [EILSEQ] shall be
 24134 stored in *errno* and the conversion state is undefined.

24135 **ERRORS**24136 The *mbrlen()* function may fail if:24137 [EINVAL] *ps* points to an object that contains an invalid conversion state.

24138 [EILSEQ] Invalid character sequence is detected.

24139 **EXAMPLES**

24140 None.

24141 **APPLICATION USAGE**

24142 None.

24143 **RATIONALE**

24144 None.

24145 **FUTURE DIRECTIONS**

24146 None.

24147 **SEE ALSO**

24148 *mbstowc()*, *mbtowc()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<wchar.h>**

24149 **CHANGE HISTORY**

24150 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
24151 (E).

24152 **Issue 6**

24153 The *mbrlen()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

24154 **NAME**

24155 mbrtowc — convert a character to a wide-character code (restartable)

24156 **SYNOPSIS**

24157 #include <wchar.h>

```
24158       size_t mbrtowc(wchar_t *restrict pwc, const char *restrict s,
24159                      size_t n, mbstate_t *restrict ps);
```

24160 **DESCRIPTION**

24161 cx The functionality described on this reference page is aligned with the ISO C standard. Any
 24162 conflict between the requirements described here and the ISO C standard is unintentional. This
 24163 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

24164 If *s* is a null pointer, the *mbrtowc()* function shall be equivalent to the call:

```
24165     mbrtowc(NULL, "", 1, ps)
```

24166 In this case, the values of the arguments *pwc* and *n* are ignored.

24167 If *s* is not a null pointer, the *mbrtowc()* function shall inspect at most *n* bytes beginning at the
 24168 byte pointed to by *s* to determine the number of bytes needed to complete the next character
 24169 (including any shift sequences). If the function determines that the next character is completed, it
 24170 shall determine the value of the corresponding wide character and then, if *pwc* is not a null
 24171 pointer, shall store that value in the object pointed to by *pwc*. If the corresponding wide
 24172 character is the null wide character, the resulting state described shall be the initial conversion
 24173 state.

24174 If *ps* is a null pointer, the *mbrtowc()* function shall use its own internal **mbstate_t** object, which
 24175 shall be initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t**
 24176 object pointed to by *ps* shall be used to completely describe the current conversion state of the
 24177 associated character sequence. The implementation shall behave as if no function defined in this
 24178 volume of IEEE Std 1003.1-2001 calls *mbrtowc()*.

24179 The behavior of this function is affected by the *LC_CTYPE* category of the current locale.

24180 **RETURN VALUE**

24181 The *mbrtowc()* function shall return the first of the following that applies:

24182 0 If the next *n* or fewer bytes complete the character that corresponds to the null
 24183 wide character (which is the value stored).

24184 between 1 and *n* inclusive

24185 If the next *n* or fewer bytes complete a valid character (which is the value
 24186 stored); the value returned shall be the number of bytes that complete the
 24187 character.

24188 (**size_t**)−2 If the next *n* bytes contribute to an incomplete but potentially valid character,
 24189 and all *n* bytes have been processed (no value is stored). When *n* has at least
 24190 the value of the {MB_CUR_MAX} macro, this case can only occur if *s* points at
 24191 a sequence of redundant shift sequences (for implementations with state-
 24192 dependent encodings).

24193 (**size_t**)−1 If an encoding error occurs, in which case the next *n* or fewer bytes do not
 24194 contribute to a complete and valid character (no value is stored). In this case,
 24195 [EILSEQ] shall be stored in *errno* and the conversion state is undefined.

24196 **ERRORS**

24197 The *mbrtowc()* function may fail if:

24198 CX [EINVAL] *ps* points to an object that contains an invalid conversion state.

24199 [EILSEQ] Invalid character sequence is detected.

24200 **EXAMPLES**

24201 None.

24202 **APPLICATION USAGE**

24203 None.

24204 **RATIONALE**

24205 None.

24206 **FUTURE DIRECTIONS**

24207 None.

24208 **SEE ALSO**

24209 *mbstowc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

24210 **CHANGE HISTORY**

24211 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
24212 (E).

24213 **Issue 6**

24214 The *mbrtowc()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

24215 The following new requirements on POSIX implementations derive from alignment with the
24216 Single UNIX Specification:

- 24217 • The [EINVAL] error condition is added.

24218 ISO/IEC 9899:1999 standard, Technical Corrigendum No. 1 is incorporated.

24219 **NAME**

24220 mbsinit — determine conversion object status

24221 **SYNOPSIS**

24222 #include <wchar.h>

24223 int mbsinit(const mbstate_t *ps);

24224 **DESCRIPTION**

24225 cx The functionality described on this reference page is aligned with the ISO C standard. Any
 24226 conflict between the requirements described here and the ISO C standard is unintentional. This
 24227 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

24228 If *ps* is not a null pointer, the *mbsinit()* function shall determine whether the object pointed to by
 24229 *ps* describes an initial conversion state.

24230 **RETURN VALUE**

24231 The *mbsinit()* function shall return non-zero if *ps* is a null pointer, or if the pointed-to object
 24232 describes an initial conversion state; otherwise, it shall return zero.

24233 If an **mbstate_t** object is altered by any of the functions described as “restartable”, and is then
 24234 used with a different character sequence, or in the other conversion direction, or with a different
 24235 *LC_CTYPE* category setting than on earlier function calls, the behavior is undefined.

24236 **ERRORS**

24237 No errors are defined.

24238 **EXAMPLES**

24239 None.

24240 **APPLICATION USAGE**

24241 The **mbstate_t** object is used to describe the current conversion state from a particular character
 24242 sequence to a wide-character sequence (or *vice versa*) under the rules of a particular setting of the
 24243 *LC_CTYPE* category of the current locale.

24244 The initial conversion state corresponds, for a conversion in either direction, to the beginning of
 24245 a new character sequence in the initial shift state. A zero valued **mbstate_t** object is at least one
 24246 way to describe an initial conversion state. A zero valued **mbstate_t** object can be used to initiate
 24247 conversion involving any character sequence, in any *LC_CTYPE* category setting.

24248 **RATIONALE**

24249 None.

24250 **FUTURE DIRECTIONS**

24251 None.

24252 **SEE ALSO**

24253 *mbrlen()*, *mbrtowc()*, *wcrtomb()*, *mbsrtowcs()*, *wcsrtombs()*, the Base Definitions volume of
 24254 IEEE Std 1003.1-2001, <wchar.h>

24255 **CHANGE HISTORY**

24256 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
 24257 (E).

24258 NAME

24259 mbsrtowcs — convert a character string to a wide-character string (restartable)

24260 SYNOPSIS

24261 #include <wchar.h>

24262 size_t mbsrtowcs(wchar_t *restrict dst, const char **restrict src,
24263 size_t len, mbstate_t *restrict ps);

24264 DESCRIPTION

24265 CX The functionality described on this reference page is aligned with the ISO C standard. Any
24266 conflict between the requirements described here and the ISO C standard is unintentional. This
24267 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

24268 The *mbsrtowcs()* function shall convert a sequence of characters, beginning in the conversion
24269 state described by the object pointed to by *ps*, from the array indirectly pointed to by *src* into a
24270 sequence of corresponding wide characters. If *dst* is not a null pointer, the converted characters
24271 shall be stored into the array pointed to by *dst*. Conversion continues up to and including a
24272 terminating null character, which shall also be stored. Conversion shall stop early in either of the
24273 following cases:

- 24274 • A sequence of bytes is encountered that does not form a valid character.
- 24275 • *len* codes have been stored into the array pointed to by *dst* (and *dst* is not a null pointer).

24276 Each conversion shall take place as if by a call to the *mbrtowc()* function.

24277 If *dst* is not a null pointer, the pointer object pointed to by *src* shall be assigned either a null
24278 pointer (if conversion stopped due to reaching a terminating null character) or the address just
24279 past the last character converted (if any). If conversion stopped due to reaching a terminating
24280 null character, and if *dst* is not a null pointer, the resulting state described shall be the initial
24281 conversion state.

24282 If *ps* is a null pointer, the *mbsrtowcs()* function shall use its own internal **mbstate_t** object, which
24283 is initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t** object
24284 pointed to by *ps* shall be used to completely describe the current conversion state of the
24285 associated character sequence. The implementation behaves as if no function defined in this
24286 volume of IEEE Std 1003.1-2001 calls *mbsrtowcs()*.

24287 The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

24288 RETURN VALUE

24289 If the input conversion encounters a sequence of bytes that do not form a valid character, an
24290 encoding error occurs. In this case, the *mbsrtowcs()* function stores the value of the macro
24291 [EILSEQ] in *errno* and shall return (**size_t**)-1; the conversion state is undefined. Otherwise, it
24292 shall return the number of characters successfully converted, not including the terminating null
24293 (if any).

24294 ERRORS

24295 The *mbsrtowcs()* function may fail if:

- 24296 CX [EINVAL] *ps* points to an object that contains an invalid conversion state.
- 24297 [EILSEQ] Invalid character sequence is detected.

24298 **EXAMPLES**

24299 None.

24300 **APPLICATION USAGE**

24301 None.

24302 **RATIONALE**

24303 None.

24304 **FUTURE DIRECTIONS**

24305 None.

24306 **SEE ALSO**24307 *mbsinit()*, *mbrtowc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>24308 **CHANGE HISTORY**24309 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
24310 (E).24311 **Issue 6**24312 The *mbsrtowcs()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

24313 The [EINVAL] error condition is marked CX.

24314 **NAME**

24315 mbstowcs — convert a character string to a wide-character string

24316 **SYNOPSIS**

24317 #include <stdlib.h>

24318 size_t mbstowcs(wchar_t *restrict pwcs, const char *restrict s,
24319 size_t n);

24320 **DESCRIPTION**

24321 CX The functionality described on this reference page is aligned with the ISO C standard. Any
24322 conflict between the requirements described here and the ISO C standard is unintentional. This
24323 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

24324 The *mbstowcs()* function shall convert a sequence of characters that begins in the initial shift
24325 state from the array pointed to by *s* into a sequence of corresponding wide-character codes and
24326 shall store not more than *n* wide-character codes into the array pointed to by *pwcs*. No
24327 characters that follow a null byte (which is converted into a wide-character code with value 0)
24328 shall be examined or converted. Each character shall be converted as if by a call to *mbtowc()*,
24329 except that the shift state of *mbtowc()* is not affected.

24330 No more than *n* elements shall be modified in the array pointed to by *pwcs*. If copying takes
24331 place between objects that overlap, the behavior is undefined.

24332 XSI The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale. If
24333 *pwcs* is a null pointer, *mbstowcs()* shall return the length required to convert the entire array
24334 regardless of the value of *n*, but no values are stored.

24335 **RETURN VALUE**

24336 CX If an invalid character is encountered, *mbstowcs()* shall return **(size_t)-1** and may set *errno* to
24337 indicate the error.

24338 XSI Otherwise, *mbstowcs()* shall return the number of the array elements modified (or required if
24339 *pwcs* is null), not including a terminating 0 code, if any. The array shall not be zero-terminated if
24340 the value returned is *n*.

24341 **ERRORS**24342 The *mbstowcs()* function may fail if:

24343 XSI [EILSEQ] Invalid byte sequence is detected.

24344 **EXAMPLES**

24345 None.

24346 **APPLICATION USAGE**

24347 None.

24348 **RATIONALE**

24349 None.

24350 **FUTURE DIRECTIONS**

24351 None.

24352 **SEE ALSO**

24353 *mblen()*, *mbtowc()*, *wctomb()*, *wcstombs()*, the Base Definitions volume of IEEE Std 1003.1-2001,
24354 <stdlib.h>

24355 **CHANGE HISTORY**

24356 First released in Issue 4. Aligned with the ISO C standard.

24357 **Issue 6**

24358 The *mbstowcs()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

24359 Extensions beyond the ISO C standard are marked.

24360 **NAME**

24361 mbtowc — convert a character to a wide-character code

24362 **SYNOPSIS**

24363 #include <stdlib.h>

24364 int mbtowc(wchar_t *restrict *pwc*, const char *restrict *s*, size_t *n*);24365 **DESCRIPTION**

24366 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 24367 conflict between the requirements described here and the ISO C standard is unintentional. This
 24368 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

24369 If *s* is not a null pointer, *mbtowc()* shall determine the number of bytes that constitute the
 24370 character pointed to by *s*. It shall then determine the wide-character code for the value of type
 24371 **wchar_t** that corresponds to that character. (The value of the wide-character code corresponding
 24372 to the null byte is 0.) If the character is valid and *pwc* is not a null pointer, *mbtowc()* shall store
 24373 the wide-character code in the object pointed to by *pwc*.

24374 The behavior of this function is affected by the *LC_CTYPE* category of the current locale. For a
 24375 state-dependent encoding, this function is placed into its initial state by a call for which its
 24376 character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null
 24377 pointer shall cause the internal state of the function to be altered as necessary. A call with *s* as a
 24378 null pointer shall cause this function to return a non-zero value if encodings have state
 24379 dependency, and 0 otherwise. If the implementation employs special bytes to change the shift
 24380 state, these bytes shall not produce separate wide-character codes, but shall be grouped with an
 24381 adjacent character. Changing the *LC_CTYPE* category causes the shift state of this function to be
 24382 unspecified. At most *n* bytes of the array pointed to by *s* shall be examined.

24383 The implementation shall behave as if no function defined in this volume of
 24384 IEEE Std 1003.1-2001 calls *mbtowc()*.

24385 **RETURN VALUE**

24386 If *s* is a null pointer, *mbtowc()* shall return a non-zero or 0 value, if character encodings,
 24387 respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, *mbtowc()*
 24388 shall either return 0 (if *s* points to the null byte), or return the number of bytes that constitute the
 24389 **CX** converted character (if the next *n* or fewer bytes form a valid character), or return -1 and may
 24390 set *errno* to indicate the error (if they do not form a valid character).

24391 In no case shall the value returned be greater than *n* or the value of the {MB_CUR_MAX} macro.

24392 **ERRORS**

24393 The *mbtowc()* function may fail if:

24394 **XSI** [EILSEQ] Invalid character sequence is detected.

24395 **EXAMPLES**

24396 None.

24397 **APPLICATION USAGE**

24398 None.

24399 **RATIONALE**

24400 None.

24401 **FUTURE DIRECTIONS**

24402 None.

24403 **SEE ALSO**

24404 *mblen()*, *mbstowcs()*, *wctomb()*, *wcstombs()*, the Base Definitions volume of IEEE Std 1003.1-2001,
24405 **<stdlib.h>**

24406 **CHANGE HISTORY**

24407 First released in Issue 4. Aligned with the ISO C standard.

24408 **Issue 6**

24409 The *mbtowc()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

24410 Extensions beyond the ISO C standard are marked.

24411 **NAME**

24412 memccpy — copy bytes in memory

24413 **SYNOPSIS**24414 XSI `#include <string.h>`

```
24415 void *memccpy(void *restrict s1, const void *restrict s2,  
24416               int c, size_t n);  
24417
```

24418 **DESCRIPTION**

24419 The *memccpy()* function shall copy bytes from memory area *s2* into *s1*, stopping after the first
24420 occurrence of byte *c* (converted to an **unsigned char**) is copied, or after *n* bytes are copied,
24421 whichever comes first. If copying takes place between objects that overlap, the behavior is
24422 undefined.

24423 **RETURN VALUE**

24424 The *memccpy()* function shall return a pointer to the byte after the copy of *c* in *s1*, or a null
24425 pointer if *c* was not found in the first *n* bytes of *s2*.

24426 **ERRORS**

24427 No errors are defined.

24428 **EXAMPLES**

24429 None.

24430 **APPLICATION USAGE**24431 The *memccpy()* function does not check for the overflow of the receiving memory area.24432 **RATIONALE**

24433 None.

24434 **FUTURE DIRECTIONS**

24435 None.

24436 **SEE ALSO**24437 The Base Definitions volume of IEEE Std 1003.1-2001, **<string.h>**24438 **CHANGE HISTORY**

24439 First released in Issue 1. Derived from Issue 1 of the SVID.

24440 **Issue 6**

24441 The **restrict** keyword is added to the *memccpy()* prototype for alignment with the
24442 ISO/IEC 9899:1999 standard.

24443 **NAME**

24444 memchr — find byte in memory

24445 **SYNOPSIS**

24446 #include <string.h>

24447 void *memchr(const void *s, int c, size_t n);

24448 **DESCRIPTION**

24449 cx The functionality described on this reference page is aligned with the ISO C standard. Any
24450 conflict between the requirements described here and the ISO C standard is unintentional. This
24451 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

24452 The *memchr()* function shall locate the first occurrence of *c* (converted to an **unsigned char**) in
24453 the initial *n* bytes (each interpreted as **unsigned char**) of the object pointed to by *s*.

24454 **RETURN VALUE**

24455 The *memchr()* function shall return a pointer to the located byte, or a null pointer if the byte does
24456 not occur in the object.

24457 **ERRORS**

24458 No errors are defined.

24459 **EXAMPLES**

24460 None.

24461 **APPLICATION USAGE**

24462 None.

24463 **RATIONALE**

24464 None.

24465 **FUTURE DIRECTIONS**

24466 None.

24467 **SEE ALSO**24468 The Base Definitions volume of IEEE Std 1003.1-2001, <**string.h**>24469 **CHANGE HISTORY**

24470 First released in Issue 1. Derived from Issue 1 of the SVID.

24471 NAME

24472 memcmp — compare bytes in memory

24473 SYNOPSIS

24474 #include <string.h>

24475 int memcmp(const void *s1, const void *s2, size_t n);

24476 DESCRIPTION

24477 cx The functionality described on this reference page is aligned with the ISO C standard. Any
24478 conflict between the requirements described here and the ISO C standard is unintentional. This
24479 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

24480 The *memcmp()* function shall compare the first *n* bytes (each interpreted as **unsigned char**) of the
24481 object pointed to by *s1* to the first *n* bytes of the object pointed to by *s2*.

24482 The sign of a non-zero return value shall be determined by the sign of the difference between the
24483 values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the objects
24484 being compared.

24485 RETURN VALUE

24486 The *memcmp()* function shall return an integer greater than, equal to, or less than 0, if the object
24487 pointed to by *s1* is greater than, equal to, or less than the object pointed to by *s2*, respectively.

24488 ERRORS

24489 No errors are defined.

24490 EXAMPLES

24491 None.

24492 APPLICATION USAGE

24493 None.

24494 RATIONALE

24495 None.

24496 FUTURE DIRECTIONS

24497 None.

24498 SEE ALSO

24499 The Base Definitions volume of IEEE Std 1003.1-2001, <**string.h**>

24500 CHANGE HISTORY

24501 First released in Issue 1. Derived from Issue 1 of the SVID.

24502 **NAME**

24503 memcpy — copy bytes in memory

24504 **SYNOPSIS**

24505 #include <string.h>

24506 void *memcpy(void *restrict *s1*, const void *restrict *s2*, size_t *n*);24507 **DESCRIPTION**

24508 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
24509 conflict between the requirements described here and the ISO C standard is unintentional. This
24510 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

24511 The *memcpy()* function shall copy *n* bytes from the object pointed to by *s2* into the object pointed
24512 to by *s1*. If copying takes place between objects that overlap, the behavior is undefined.

24513 **RETURN VALUE**24514 The *memcpy()* function shall return *s1*; no return value is reserved to indicate an error.24515 **ERRORS**

24516 No errors are defined.

24517 **EXAMPLES**

24518 None.

24519 **APPLICATION USAGE**24520 The *memcpy()* function does not check for the overflow of the receiving memory area.24521 **RATIONALE**

24522 None.

24523 **FUTURE DIRECTIONS**

24524 None.

24525 **SEE ALSO**24526 The Base Definitions volume of IEEE Std 1003.1-2001, <**string.h**>24527 **CHANGE HISTORY**

24528 First released in Issue 1. Derived from Issue 1 of the SVID.

24529 **Issue 6**24530 The *memcpy()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

24531 **NAME**

24532 memmove — copy bytes in memory with overlapping areas

24533 **SYNOPSIS**

24534 #include <string.h>

24535 void *memmove(void *s1, const void *s2, size_t n);

24536 **DESCRIPTION**

24537 cx The functionality described on this reference page is aligned with the ISO C standard. Any
24538 conflict between the requirements described here and the ISO C standard is unintentional. This
24539 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

24540 The *memmove()* function shall copy *n* bytes from the object pointed to by *s2* into the object
24541 pointed to by *s1*. Copying takes place as if the *n* bytes from the object pointed to by *s2* are first
24542 copied into a temporary array of *n* bytes that does not overlap the objects pointed to by *s1* and
24543 *s2*, and then the *n* bytes from the temporary array are copied into the object pointed to by *s1*.

24544 **RETURN VALUE**24545 The *memmove()* function shall return *s1*; no return value is reserved to indicate an error.24546 **ERRORS**

24547 No errors are defined.

24548 **EXAMPLES**

24549 None.

24550 **APPLICATION USAGE**

24551 None.

24552 **RATIONALE**

24553 None.

24554 **FUTURE DIRECTIONS**

24555 None.

24556 **SEE ALSO**24557 The Base Definitions volume of IEEE Std 1003.1-2001, <**string.h**>24558 **CHANGE HISTORY**

24559 First released in Issue 4. Derived from the ANSI C standard.

24560 **NAME**

24561 memset — set bytes in memory

24562 **SYNOPSIS**

24563 #include <string.h>

24564 void *memset(void *s, int c, size_t n);

24565 **DESCRIPTION**

24566 cx The functionality described on this reference page is aligned with the ISO C standard. Any
24567 conflict between the requirements described here and the ISO C standard is unintentional. This
24568 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

24569 The *memset()* function shall copy *c* (converted to an **unsigned char**) into each of the first *n* bytes
24570 of the object pointed to by *s*.

24571 **RETURN VALUE**24572 The *memset()* function shall return *s*; no return value is reserved to indicate an error.24573 **ERRORS**

24574 No errors are defined.

24575 **EXAMPLES**

24576 None.

24577 **APPLICATION USAGE**

24578 None.

24579 **RATIONALE**

24580 None.

24581 **FUTURE DIRECTIONS**

24582 None.

24583 **SEE ALSO**24584 The Base Definitions volume of IEEE Std 1003.1-2001, <**string.h**>24585 **CHANGE HISTORY**

24586 First released in Issue 1. Derived from Issue 1 of the SVID.

24587 NAME

24588 mkdir — make a directory

24589 SYNOPSIS

24590 #include <sys/stat.h>

24591 int mkdir(const char *path, mode_t mode);

24592 DESCRIPTION

24593 The *mkdir()* function shall create a new directory with name *path*. The file permission bits of the
24594 new directory shall be initialized from *mode*. These file permission bits of the *mode* argument
24595 shall be modified by the process' file creation mask.

24596 When bits in *mode* other than the file permission bits are set, the meaning of these additional bits
24597 is implementation-defined.

24598 The directory's user ID shall be set to the process' effective user ID. The directory's group ID
24599 shall be set to the group ID of the parent directory or to the effective group ID of the process.
24600 Implementations shall provide a way to initialize the directory's group ID to the group ID of the
24601 parent directory. Implementations may, but need not, provide an implementation-defined way
24602 to initialize the directory's group ID to the effective group ID of the calling process.

24603 The newly created directory shall be an empty directory.

24604 If *path* names a symbolic link, *mkdir()* shall fail and set *errno* to [EEXIST].

24605 Upon successful completion, *mkdir()* shall mark for update the *st_atime*, *st_ctime*, and *st_mtime*
24606 fields of the directory. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the
24607 new entry shall be marked for update.

24608 RETURN VALUE

24609 Upon successful completion, *mkdir()* shall return 0. Otherwise, -1 shall be returned, no directory
24610 shall be created, and *errno* shall be set to indicate the error.

24611 ERRORS

24612 The *mkdir()* function shall fail if:

24613 [EACCES] Search permission is denied on a component of the path prefix, or write
24614 permission is denied on the parent directory of the directory to be created.

24615 [EEXIST] The named file exists.

24616 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
24617 argument.

24618 [EMLINK] The link count of the parent directory would exceed {LINK_MAX}.

24619 [ENAMETOOLONG]

24620 The length of the *path* argument exceeds {PATH_MAX} or a pathname
24621 component is longer than {NAME_MAX}.

24622 [ENOENT] A component of the path prefix specified by *path* does not name an existing
24623 directory or *path* is an empty string.

24624 [ENOSPC] The file system does not contain enough space to hold the contents of the new
24625 directory or to extend the parent directory of the new directory.

24626 [ENOTDIR] A component of the path prefix is not a directory.

24627 [EROFS] The parent directory resides on a read-only file system.

24628 The *mkdir()* function may fail if:

24629 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
24630 resolution of the *path* argument.

24631 [ENAMETOOLONG]

24632 As a result of encountering a symbolic link in resolution of the *path* argument,
24633 the length of the substituted pathname string exceeded {PATH_MAX}.

24634 EXAMPLES

24635 Creating a Directory

24636 The following example shows how to create a directory named */home/cnd/mod1*, with
24637 read/write/search permissions for owner and group, and with read/search permissions for
24638 others.

```
24639 #include <sys/types.h>
```

```
24640 #include <sys/stat.h>
```

```
24641 int status;
```

```
24642 ...
```

```
24643 status = mkdir("/home/cnd/mod1", S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH);
```

24644 APPLICATION USAGE

24645 None.

24646 RATIONALE

24647 The *mkdir()* function originated in 4.2 BSD and was added to System V in Release 3.0.

24648 4.3 BSD detects [ENAMETOOLONG].

24649 The POSIX.1-1990 standard required that the group ID of a newly created directory be set to the
24650 group ID of its parent directory or to the effective group ID of the creating process. FIPS 151-2
24651 required that implementations provide a way to have the group ID be set to the group ID of the
24652 containing directory, but did not prohibit implementations also supporting a way to set the
24653 group ID to the effective group ID of the creating process. Conforming applications should not
24654 assume which group ID will be used. If it matters, an application can use *chown()* to set the
24655 group ID after the directory is created, or determine under what conditions the implementation
24656 will set the desired group ID.

24657 FUTURE DIRECTIONS

24658 None.

24659 SEE ALSO

24660 *umask()*, the Base Definitions volume of IEEE Std 1003.1-2001, *<sys/stat.h>*, *<sys/types.h>*

24661 CHANGE HISTORY

24662 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

24663 Issue 6

24664 In the SYNOPSIS, the optional include of the *<sys/types.h>* header is removed.

24665 The following new requirements on POSIX implementations derive from alignment with the
24666 Single UNIX Specification:

- 24667 • The requirement to include *<sys/types.h>* has been removed. Although *<sys/types.h>* was
- 24668 required for conforming implementations of previous POSIX specifications, it was not
- 24669 required for UNIX applications.

- 24670 • The [ELOOP] mandatory error condition is added.
- 24671 • A second [ENAMETOOLONG] is added as an optional error condition.
- 24672 The following changes were made to align with the IEEE P1003.1a draft standard:
- 24673 • The [ELOOP] optional error condition is added.

24674 **NAME**

24675 mkfifo — make a FIFO special file

24676 **SYNOPSIS**

24677 #include <sys/stat.h>

24678 int mkfifo(const char *path, mode_t mode);

24679 **DESCRIPTION**

24680 The *mkfifo()* function shall create a new FIFO special file named by the pathname pointed to by
 24681 *path*. The file permission bits of the new FIFO shall be initialized from *mode*. The file permission
 24682 bits of the *mode* argument shall be modified by the process' file creation mask.

24683 When bits in *mode* other than the file permission bits are set, the effect is implementation-
 24684 defined.

24685 If *path* names a symbolic link, *mkfifo()* shall fail and set *errno* to [EEXIST].

24686 The FIFO's user ID shall be set to the process' effective user ID. The FIFO's group ID shall be set
 24687 to the group ID of the parent directory or to the effective group ID of the process.
 24688 Implementations shall provide a way to initialize the FIFO's group ID to the group ID of the
 24689 parent directory. Implementations may, but need not, provide an implementation-defined way
 24690 to initialize the FIFO's group ID to the effective group ID of the calling process.

24691 Upon successful completion, *mkfifo()* shall mark for update the *st_atime*, *st_ctime*, and *st_mtime*
 24692 fields of the file. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the new
 24693 entry shall be marked for update.

24694 **RETURN VALUE**

24695 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned, no FIFO shall
 24696 be created, and *errno* shall be set to indicate the error.

24697 **ERRORS**24698 The *mkfifo()* function shall fail if:

24699 [EACCES] A component of the path prefix denies search permission, or write permission
 24700 is denied on the parent directory of the FIFO to be created.

24701 [EEXIST] The named file already exists.

24702 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 24703 argument.

24704 [ENAMETOOLONG]

24705 The length of the *path* argument exceeds {PATH_MAX} or a pathname
 24706 component is longer than {NAME_MAX}.

24707 [ENOENT] A component of the path prefix specified by *path* does not name an existing
 24708 directory or *path* is an empty string.

24709 [ENOSPC] The directory that would contain the new file cannot be extended or the file
 24710 system is out of file-allocation resources.

24711 [ENOTDIR] A component of the path prefix is not a directory.

24712 [EROFS] The named file resides on a read-only file system.

24713 The *mkfifo()* function may fail if:

24714 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 24715 resolution of the *path* argument.

24716 [ENAMETOOLONG]
 24717 As a result of encountering a symbolic link in resolution of the *path* argument,
 24718 the length of the substituted pathname string exceeded {PATH_MAX}.

24719 EXAMPLES

24720 Creating a FIFO File

24721 The following example shows how to create a FIFO file named `/home/cnd/mod_done`, with
 24722 read/write permissions for owner, and with read permissions for group and others.

```
24723 #include <sys/types.h>
24724 #include <sys/stat.h>

24725 int status;
24726 ...
24727 status = mkfifo("/home/cnd/mod_done", S_IWUSR | S_IRUSR |
24728               S_IRGRP | S_IROTH);
```

24729 APPLICATION USAGE

24730 None.

24731 RATIONALE

24732 The syntax of this function is intended to maintain compatibility with historical
 24733 implementations of *mknod()*. The latter function was included in the 1984 /usr/group standard
 24734 but only for use in creating FIFO special files. The *mknod()* function was originally excluded
 24735 from the POSIX.1-1988 standard as implementation-defined and replaced by *mkdir()* and
 24736 *mkfifo()*. The *mknod()* function is now included for alignment with the Single UNIX
 24737 Specification.

24738 The POSIX.1-1990 standard required that the group ID of a newly created FIFO be set to the
 24739 group ID of its parent directory or to the effective group ID of the creating process. FIPS 151-2
 24740 required that implementations provide a way to have the group ID be set to the group ID of the
 24741 containing directory, but did not prohibit implementations also supporting a way to set the
 24742 group ID to the effective group ID of the creating process. Conforming applications should not
 24743 assume which group ID will be used. If it matters, an application can use *chown()* to set the
 24744 group ID after the FIFO is created, or determine under what conditions the implementation will
 24745 set the desired group ID.

24746 FUTURE DIRECTIONS

24747 None.

24748 SEE ALSO

24749 *umask()*, the Base Definitions volume of IEEE Std 1003.1-2001, `<sys/stat.h>`, `<sys/types.h>`

24750 CHANGE HISTORY

24751 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

24752 Issue 6

24753 In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

24754 The following new requirements on POSIX implementations derive from alignment with the
 24755 Single UNIX Specification:

- 24756 • The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was
 24757 required for conforming implementations of previous POSIX specifications, it was not
 24758 required for UNIX applications.

- 24759 • The [ELOOP] mandatory error condition is added.
- 24760 • A second [ENAMETOOLONG] is added as an optional error condition.
- 24761 The following changes were made to align with the IEEE P1003.1a draft standard:
- 24762 • The [ELOOP] optional error condition is added.

24763 **NAME**24764 **mknod** — make a directory, a special file, or a regular file24765 **SYNOPSIS**24766 xSI `#include <sys/stat.h>`24767 `int mknod(const char *path, mode_t mode, dev_t dev);`

24768

24769 **DESCRIPTION**24770 The *mknod()* function shall create a new file named by the pathname to which the argument *path*
24771 points.24772 The file type for *path* is OR'ed into the *mode* argument, and the application shall select one of the
24773 following symbolic constants:

24774

24775

24776

24777

24778

24779

24780

Name	Description
S_IFIFO	FIFO-special
S_IFCHR	Character-special (non-portable)
S_IFDIR	Directory (non-portable)
S_IFBLK	Block-special (non-portable)
S_IFREG	Regular (non-portable)

24781 The only portable use of *mknod()* is to create a FIFO-special file. If *mode* is not S_IFIFO or *dev* is
24782 not 0, the behavior of *mknod()* is unspecified.24783 The permissions for the new file are OR'ed into the *mode* argument, and may be selected from
24784 any combination of the following symbolic constants:

24785

24786

24787

24788

24789

24790

24791

24792

24793

24794

24795

24796

24797

24798

24799

24800

24801

Name	Description
S_ISUID	Set user ID on execution.
S_ISGID	Set group ID on execution.
S_IRWXU	Read, write, or execute (search) by owner.
S_IRUSR	Read by owner.
S_IWUSR	Write by owner.
S_IXUSR	Execute (search) by owner.
S_IRWXG	Read, write, or execute (search) by group.
S_IRGRP	Read by group.
S_IWGRP	Write by group.
S_IXGRP	Execute (search) by group.
S_IRWXO	Read, write, or execute (search) by others.
S_IROTH	Read by others.
S_IWOTH	Write by others.
S_IXOTH	Execute (search) by others.
S_ISVTX	On directories, restricted deletion flag.

24802 The user ID of the file shall be initialized to the effective user ID of the process. The group ID of
 24803 the file shall be initialized to either the effective group ID of the process or the group ID of the
 24804 parent directory. Implementations shall provide a way to initialize the file's group ID to the
 24805 group ID of the parent directory. Implementations may, but need not, provide an
 24806 implementation-defined way to initialize the file's group ID to the effective group ID of the
 24807 calling process. The owner, group, and other permission bits of *mode* shall be modified by the file
 24808 mode creation mask of the process. The *mknod()* function shall clear each bit whose
 24809 corresponding bit in the file mode creation mask of the process is set.

- 24810 If *path* names a symbolic link, *mknod()* shall fail and set *errno* to [EEXIST].
- 24811 Upon successful completion, *mknod()* shall mark for update the *st_atime*, *st_ctime*, and *st_mtime*
 24812 fields of the file. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the new
 24813 entry shall be marked for update.
- 24814 Only a process with appropriate privileges may invoke *mknod()* for file types other than FIFO-
 24815 special.
- 24816 **RETURN VALUE**
- 24817 Upon successful completion, *mknod()* shall return 0. Otherwise, it shall return -1, the new file
 24818 shall not be created, and *errno* shall be set to indicate the error.
- 24819 **ERRORS**
- 24820 The *mknod()* function shall fail if:
- 24821 [EACCES] A component of the path prefix denies search permission, or write permission
 24822 is denied on the parent directory.
- 24823 [EEXIST] The named file exists.
- 24824 [EINVAL] An invalid argument exists.
- 24825 [EIO] An I/O error occurred while accessing the file system.
- 24826 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 24827 argument.
- 24828 [ENAMETOOLONG]
 24829 The length of a pathname exceeds {PATH_MAX} or a pathname component is
 24830 longer than {NAME_MAX}.
- 24831 [ENOENT] A component of the path prefix specified by *path* does not name an existing
 24832 directory or *path* is an empty string.
- 24833 [ENOSPC] The directory that would contain the new file cannot be extended or the file
 24834 system is out of file allocation resources.
- 24835 [ENOTDIR] A component of the path prefix is not a directory.
- 24836 [EPERM] The invoking process does not have appropriate privileges and the file type is
 24837 not FIFO-special.
- 24838 [EROFS] The directory in which the file is to be created is located on a read-only file
 24839 system.
- 24840 The *mknod()* function may fail if:
- 24841 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 24842 resolution of the *path* argument.
- 24843 [ENAMETOOLONG]
 24844 Pathname resolution of a symbolic link produced an intermediate result
 24845 whose length exceeds {PATH_MAX}.

24846 **EXAMPLES**24847 **Creating a FIFO Special File**

24848 The following example shows how to create a FIFO special file named `/home/cnd/mod_done`,
 24849 with read/write permissions for owner, and with read permissions for group and others.

```
24850 #include <sys/types.h>
24851 #include <sys/stat.h>

24852 dev_t dev;
24853 int status;
24854 ...
24855 status = mknod("/home/cnd/mod_done", S_IFIFO | S_IWUSR |
24856               S_IRUSR | S_IRGRP | S_IROTH, dev);
```

24857 **APPLICATION USAGE**

24858 The `mkfifo()` function is preferred over this function for making FIFO special files.

24859 **RATIONALE**

24860 The POSIX.1-1990 standard required that the group ID of a newly created file be set to the group
 24861 ID of its parent directory or to the effective group ID of the creating process. FIPS 151-2 required
 24862 that implementations provide a way to have the group ID be set to the group ID of the
 24863 containing directory, but did not prohibit implementations also supporting a way to set the
 24864 group ID to the effective group ID of the creating process. Conforming applications should not
 24865 assume which group ID will be used. If it matters, an application can use `chown()` to set the
 24866 group ID after the file is created, or determine under what conditions the implementation will
 24867 set the desired group ID.

24868 **FUTURE DIRECTIONS**

24869 None.

24870 **SEE ALSO**

24871 `chmod()`, `creat()`, `exec`, `mkdir()`, `mkfifo()`, `open()`, `stat()`, `umask()`, the Base Definitions volume of
 24872 IEEE Std 1003.1-2001, `<sys/stat.h>`

24873 **CHANGE HISTORY**

24874 First released in Issue 4, Version 2.

24875 **Issue 5**

24876 Moved from X/OPEN UNIX extension to BASE.

24877 **Issue 6**

24878 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

24879 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
 24880 [ELOOP] error condition is added.

24881 **NAME**

24882 mkstemp — make a unique filename

24883 **SYNOPSIS**24884 XSI `#include <stdlib.h>`24885 `int mkstemp(char *template);`

24886

24887 **DESCRIPTION**

24888 The *mkstemp()* function shall replace the contents of the string pointed to by *template* by a unique
 24889 filename, and return a file descriptor for the file open for reading and writing. The function thus
 24890 prevents any possible race condition between testing whether the file exists and opening it for
 24891 use. The string in *template* should look like a filename with six trailing 'X's; *mkstemp()* replaces
 24892 each 'X' with a character from the portable filename character set. The characters are chosen
 24893 such that the resulting name does not duplicate the name of an existing file at the time of a call
 24894 to *mkstemp()*.

24895 **RETURN VALUE**

24896 Upon successful completion, *mkstemp()* shall return an open file descriptor. Otherwise, -1 shall
 24897 be returned if no suitable file could be created.

24898 **ERRORS**

24899 No errors are defined.

24900 **EXAMPLES**24901 **Generating a Filename**

24902 The following example creates a file with a 10-character name beginning with the characters
 24903 "file" and opens the file for reading and writing. The value returned as the value of *fd* is a file
 24904 descriptor that identifies the file.

```
24905 #include <stdlib.h>
24906 ...
24907 char template[] = "/tmp/fileXXXXXX";
24908 int fd;
24909 fd = mkstemp(template);
```

24910 **APPLICATION USAGE**

24911 It is possible to run out of letters.

24912 The *mkstemp()* function need not check to determine whether the filename part of *template*
 24913 exceeds the maximum allowable filename length.

24914 **RATIONALE**

24915 None.

24916 **FUTURE DIRECTIONS**

24917 None.

24918 **SEE ALSO**

24919 *getpid()*, *open()*, *tmpfile()*, *tmpnam()*, the Base Definitions volume of IEEE Std 1003.1-2001,
 24920 `<stdlib.h>`

24921 **CHANGE HISTORY**

24922 First released in Issue 4, Version 2.

24923 **Issue 5**

24924 Moved from X/OPEN UNIX extension to BASE.

24925 **NAME**24926 mktemp — make a unique filename (**LEGACY**)24927 **SYNOPSIS**24928 XSI `#include <stdlib.h>`24929 `char *mktemp(char *template);`

24930

24931 **DESCRIPTION**

24932 The *mktemp()* function shall replace the contents of the string pointed to by *template* by a unique
 24933 filename and return *template*. The application shall initialize *template* to be a filename with six
 24934 trailing 'x's; *mktemp()* shall replace each 'x' with a single byte character from the portable
 24935 filename character set.

24936 **RETURN VALUE**

24937 The *mktemp()* function shall return the pointer *template*. If a unique name cannot be created,
 24938 *template* shall point to a null string.

24939 **ERRORS**

24940 No errors are defined.

24941 **EXAMPLES**24942 **Generating a Filename**

24943 The following example replaces the contents of the "template" string with a 10-character
 24944 filename beginning with the characters "file" and returns a pointer to the "template" string
 24945 that contains the new filename.

```
24946 #include <stdlib.h>
24947 ...
24948 char *template = "/tmp/fileXXXXXX";
24949 char *ptr;
24950 ptr = mktemp(template);
```

24951 **APPLICATION USAGE**

24952 Between the time a pathname is created and the file opened, it is possible for some other process
 24953 to create a file with the same name. The *mkstemp()* function avoids this problem and is preferred
 24954 over this function.

24955 **RATIONALE**

24956 None.

24957 **FUTURE DIRECTIONS**

24958 This function may be withdrawn in a future version.

24959 **SEE ALSO**24960 *mkstemp()*, *tmpfile()*, *tmpnam()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>24961 **CHANGE HISTORY**

24962 First released in Issue 4, Version 2.

24963 **Issue 5**

24964 Moved from X/OPEN UNIX extension to BASE.

24965 **Issue 6**

24966 This function is marked LEGACY.

24967 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

24968 **NAME**

24969 mktime — convert broken-down time into time since the Epoch

24970 **SYNOPSIS**

24971 #include <time.h>

24972 time_t mktime(struct tm *timeptr);

24973 **DESCRIPTION**

24974 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 24975 conflict between the requirements described here and the ISO C standard is unintentional. This
 24976 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

24977 The *mktime()* function shall convert the broken-down time, expressed as local time, in the
 24978 structure pointed to by *timeptr*, into a time since the Epoch value with the same encoding as that
 24979 of the values returned by *time()*. The original values of the *tm_wday* and *tm_yday* components of
 24980 the structure are ignored, and the original values of the other components are not restricted to
 24981 the ranges described in <**time.h**>.

24982 **CX** A positive or 0 value for *tm_isdst* shall cause *mktime()* to presume initially that Daylight Savings
 24983 Time, respectively, is or is not in effect for the specified time. A negative value for *tm_isdst* shall
 24984 cause *mktime()* to attempt to determine whether Daylight Savings Time is in effect for the
 24985 specified time.

24986 Local timezone information shall be set as though *mktime()* called *tzset()*.

24987 The relationship between the **tm** structure (defined in the <**time.h**> header) and the time in
 24988 seconds since the Epoch is that the result shall be as specified in the expression given in the
 24989 definition of seconds since the Epoch (see the Base Definitions volume of IEEE Std 1003.1-2001,
 24990 Section 4.14, Seconds Since the Epoch) corrected for timezone and any seasonal time
 24991 adjustments, where the names in the structure and in the expression correspond.

24992 Upon successful completion, the values of the *tm_wday* and *tm_yday* components of the structure
 24993 shall be set appropriately, and the other components are set to represent the specified time since
 24994 the Epoch, but with their values forced to the ranges indicated in the <**time.h**> entry; the final
 24995 value of *tm_mday* shall not be set until *tm_mon* and *tm_year* are determined.

24996 **RETURN VALUE**

24997 The *mktime()* function shall return the specified time since the Epoch encoded as a value of type
 24998 **time_t**. If the time since the Epoch cannot be represented, the function shall return the value
 24999 (**time_t**)-1.

25000 **ERRORS**

25001 No errors are defined.

25002 **EXAMPLES**

25003 What day of the week is July 4, 2001?

25004 #include <stdio.h>

25005 #include <time.h>

25006 struct tm time_str;

25007 char daybuf[20];

25008 int main(void)

25009 {

25010 time_str.tm_year = 2001 - 1900;

25011 time_str.tm_mon = 7 - 1;

25012 time_str.tm_mday = 4;


```
25013         time_str.tm_hour = 0;
25014         time_str.tm_min = 0;
25015         time_str.tm_sec = 1;
25016         time_str.tm_isdst = -1;
25017         if (mktime(&time_str) == -1)
25018             (void)puts("-unknown-");
25019         else {
25020             (void)strftime(daybuf, sizeof(daybuf), "%A", &time_str);
25021             (void)puts(daybuf);
25022         }
25023         return 0;
25024     }
```

25025 APPLICATION USAGE

25026 None.

25027 RATIONALE

25028 None.

25029 FUTURE DIRECTIONS

25030 None.

25031 SEE ALSO

25032 *asctime()*, *clock()*, *ctime()*, *difftime()*, *gmtime()*, *localtime()*, *strftime()*, *strptime()*, *time()*, *utime()*,
25033 the Base Definitions volume of IEEE Std 1003.1-2001, <**time.h**>

25034 CHANGE HISTORY

25035 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard and the ANSI C
25036 standard.

25037 Issue 6

25038 Extensions beyond the ISO C standard are marked.

25039 **NAME**25040 mlock, munlock — lock or unlock a range of process address space (**REALTIME**)25041 **SYNOPSIS**25042 MLR `#include <sys/mman.h>`25043 `int mlock(const void *addr, size_t len);`25044 `int munlock(const void *addr, size_t len);`

25045

25046 **DESCRIPTION**

25047 The *mlock()* function shall cause those whole pages containing any part of the address space of
 25048 the process starting at address *addr* and continuing for *len* bytes to be memory-resident until
 25049 unlocked or until the process exits or *execs* another process image. The implementation may
 25050 require that *addr* be a multiple of {PAGESIZE}.

25051 The *munlock()* function shall unlock those whole pages containing any part of the address space
 25052 of the process starting at address *addr* and continuing for *len* bytes, regardless of how many
 25053 times *mlock()* has been called by the process for any of the pages in the specified range. The
 25054 implementation may require that *addr* be a multiple of {PAGESIZE}.

25055 If any of the pages in the range specified to a call to *munlock()* are also mapped into the address
 25056 spaces of other processes, any locks established on those pages by another process are
 25057 unaffected by the call of this process to *munlock()*. If any of the pages in the range specified by a
 25058 call to *munlock()* are also mapped into other portions of the address space of the calling process
 25059 outside the range specified, any locks established on those pages via the other mappings are also
 25060 unaffected by this call.

25061 Upon successful return from *mlock()*, pages in the specified range shall be locked and memory-
 25062 resident. Upon successful return from *munlock()*, pages in the specified range shall be unlocked
 25063 with respect to the address space of the process. Memory residency of unlocked pages is
 25064 unspecified.

25065 The appropriate privilege is required to lock process memory with *mlock()*.

25066 **RETURN VALUE**

25067 Upon successful completion, the *mlock()* and *munlock()* functions shall return a value of zero.
 25068 Otherwise, no change is made to any locks in the address space of the process, and the function
 25069 shall return a value of -1 and set *errno* to indicate the error.

25070 **ERRORS**

25071 The *mlock()* and *munlock()* functions shall fail if:

25072 [ENOMEM] Some or all of the address range specified by the *addr* and *len* arguments does
 25073 not correspond to valid mapped pages in the address space of the process.

25074 The *mlock()* function shall fail if:

25075 [EAGAIN] Some or all of the memory identified by the operation could not be locked
 25076 when the call was made.

25077 The *mlock()* and *munlock()* functions may fail if:

25078 [EINVAL] The *addr* argument is not a multiple of {PAGESIZE}.

25079 The *mlock()* function may fail if:

25080 [ENOMEM] Locking the pages mapped by the specified range would exceed an
 25081 implementation-defined limit on the amount of memory that the process may
 25082 lock.

25083 [EPERM] The calling process does not have the appropriate privilege to perform the
25084 requested operation.

25085 **EXAMPLES**

25086 None.

25087 **APPLICATION USAGE**

25088 None.

25089 **RATIONALE**

25090 None.

25091 **FUTURE DIRECTIONS**

25092 None.

25093 **SEE ALSO**

25094 *exec*, *exit()*, *fork()*, *mlockall()*, *munmap()*, the Base Definitions volume of IEEE Std 1003.1-2001,
25095 **<sys/mman.h>**

25096 **CHANGE HISTORY**

25097 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

25098 **Issue 6**

25099 The *mlock()* and *munlock()* functions are marked as part of the Range Memory Locking option.

25100 The [ENOSYS] error condition has been removed as stubs need not be provided if an
25101 implementation does not support the Range Memory Locking option.

25102 **NAME**25103 mlockall, munlockall — lock/unlock the address space of a process (**REALTIME**)25104 **SYNOPSIS**

25105 ML #include <sys/mman.h>

25106 int mlockall(int flags);

25107 int munlockall(void);

25108

25109 **DESCRIPTION**

25110 The *mlockall()* function shall cause all of the pages mapped by the address space of a process to
 25111 be memory-resident until unlocked or until the process exits or *execs* another process image. The
 25112 *flags* argument determines whether the pages to be locked are those currently mapped by the
 25113 address space of the process, those that are mapped in the future, or both. The *flags* argument is
 25114 constructed from the bitwise-inclusive OR of one or more of the following symbolic constants,
 25115 defined in <sys/mman.h>:

25116 MCL_CURRENT Lock all of the pages currently mapped into the address space of the process.

25117 MCL_FUTURE Lock all of the pages that become mapped into the address space of the
 25118 process in the future, when those mappings are established.

25119 If MCL_FUTURE is specified, and the automatic locking of future mappings eventually causes
 25120 the amount of locked memory to exceed the amount of available physical memory or any other
 25121 implementation-defined limit, the behavior is implementation-defined. The manner in which the
 25122 implementation informs the application of these situations is also implementation-defined.

25123 The *munlockall()* function shall unlock all currently mapped pages of the address space of the
 25124 process. Any pages that become mapped into the address space of the process after a call to
 25125 *munlockall()* shall not be locked, unless there is an intervening call to *mlockall()* specifying
 25126 MCL_FUTURE or a subsequent call to *mlockall()* specifying MCL_CURRENT. If pages mapped
 25127 into the address space of the process are also mapped into the address spaces of other processes
 25128 and are locked by those processes, the locks established by the other processes shall be
 25129 unaffected by a call by this process to *munlockall()*.

25130 Upon successful return from the *mlockall()* function that specifies MCL_CURRENT, all currently
 25131 mapped pages of the process' address space shall be memory-resident and locked. Upon return
 25132 from the *munlockall()* function, all currently mapped pages of the process' address space shall be
 25133 unlocked with respect to the process' address space. The memory residency of unlocked pages is
 25134 unspecified.

25135 The appropriate privilege is required to lock process memory with *mlockall()*.

25136 **RETURN VALUE**

25137 Upon successful completion, the *mlockall()* function shall return a value of zero. Otherwise, no
 25138 additional memory shall be locked, and the function shall return a value of -1 and set *errno* to
 25139 indicate the error. The effect of failure of *mlockall()* on previously existing locks in the address
 25140 space is unspecified.

25141 If it is supported by the implementation, the *munlockall()* function shall always return a value of
 25142 zero. Otherwise, the function shall return a value of -1 and set *errno* to indicate the error.

25143 **ERRORS**

25144 The *mlockall()* function shall fail if:

25145 [EAGAIN] Some or all of the memory identified by the operation could not be locked
 25146 when the call was made.

25147	[EINVAL]	The <i>flags</i> argument is zero, or includes unimplemented flags.
25148		The <i>mlockall()</i> function may fail if:
25149	[ENOMEM]	Locking all of the pages currently mapped into the address space of the
25150		process would exceed an implementation-defined limit on the amount of
25151		memory that the process may lock.
25152	[EPERM]	The calling process does not have the appropriate privilege to perform the
25153		requested operation.
25154	EXAMPLES	
25155		None.
25156	APPLICATION USAGE	
25157		None.
25158	RATIONALE	
25159		None.
25160	FUTURE DIRECTIONS	
25161		None.
25162	SEE ALSO	
25163		<i>exec</i> , <i>exit()</i> , <i>fork()</i> , <i>mlock()</i> , <i>munmap()</i> , the Base Definitions volume of IEEE Std 1003.1-2001,
25164		<sys/mman.h>
25165	CHANGE HISTORY	
25166		First released in Issue 5. Included for alignment with the POSIX Realtime Extension.
25167	Issue 6	
25168		The <i>mlockall()</i> and <i>munlockall()</i> functions are marked as part of the Process Memory Locking
25169		option.
25170		The [ENOSYS] error condition has been removed as stubs need not be provided if an
25171		implementation does not support the Process Memory Locking option.

25172 **NAME**25173 **mmap** — map pages of memory25174 **SYNOPSIS**25175 MF|SHM `#include <sys/mman.h>`

```
25176 void *mmap(void *addr, size_t len, int prot, int flags,
25177            int fildes, off_t off);
25178
```

25179 **DESCRIPTION**

25180 The *mmap()* function shall establish a mapping between a process' address space and a file,
 25181 TYM shared memory object, or typed memory object. The format of the call is as follows:

```
25182 pa=mmap(addr, len, prot, flags, fildes, off);
```

25183 The *mmap()* function shall establish a mapping between the address space of the process at an
 25184 address *pa* for *len* bytes to the memory object represented by the file descriptor *fildes* at offset *off*
 25185 for *len* bytes. The value of *pa* is an implementation-defined function of the parameter *addr* and
 25186 the values of *flags*, further described below. A successful *mmap()* call shall return *pa* as its result.
 25187 The address range starting at *pa* and continuing for *len* bytes shall be legitimate for the possible
 25188 (not necessarily current) address space of the process. The range of bytes starting at *off* and
 25189 continuing for *len* bytes shall be legitimate for the possible (not necessarily current) offsets in the
 25190 TYM file, shared memory object, or typed memory object represented by *fildes*.

25191 TYM If *fildes* represents a typed memory object opened with either the
 25192 POSIX_TYPED_MEM_ALLOCATE flag or the POSIX_TYPED_MEM_ALLOCATE_CONTIG
 25193 flag, the memory object to be mapped shall be that portion of the typed memory object allocated
 25194 by the implementation as specified below. In this case, if *off* is non-zero, the behavior of *mmap()*
 25195 is undefined. If *fildes* refers to a valid typed memory object that is not accessible from the calling
 25196 process, *mmap()* shall fail.

25197 The mapping established by *mmap()* shall replace any previous mappings for those whole pages
 25198 containing any part of the address space of the process starting at *pa* and continuing for *len*
 25199 bytes.

25200 If the size of the mapped file changes after the call to *mmap()* as a result of some other operation
 25201 on the mapped file, the effect of references to portions of the mapped region that correspond to
 25202 added or removed portions of the file is unspecified.

25203 TYM The *mmap()* function shall be supported for regular files, shared memory objects, and typed
 25204 memory objects. Support for any other type of file is unspecified.

25205 The parameter *prot* determines whether read, write, execute, or some combination of accesses
 25206 are permitted to the data being mapped. The *prot* shall be either PROT_NONE or the bitwise-
 25207 inclusive OR of one or more of the other flags in the following table, defined in the
 25208 <sys/mman.h> header.

25209

25210

25211

25212

25213

25214

Symbolic Constant	Description
PROT_READ	Data can be read.
PROT_WRITE	Data can be written.
PROT_EXEC	Data can be executed.
PROT_NONE	Data cannot be accessed.

25215 If an implementation cannot support the combination of access types specified by *prot*, the call
 25216 to *mmap()* shall fail.

25217 MPR An implementation may permit accesses other than those specified by *prot*; however, if the
 25218 Memory Protection option is supported, the implementation shall not permit a write to succeed
 25219 where PROT_WRITE has not been set or shall not permit any access where PROT_NONE alone
 25220 has been set. The implementation shall support at least the following values of *prot*:
 25221 PROT_NONE, PROT_READ, PROT_WRITE, and the bitwise-inclusive OR of PROT_READ and
 25222 PROT_WRITE. If the Memory Protection option is not supported, the result of any access that
 25223 conflicts with the specified protection is undefined. The file descriptor *fildes* shall have been
 25224 opened with read permission, regardless of the protection options specified. If PROT_WRITE is
 25225 specified, the application shall ensure that it has opened the file descriptor *fildes* with write
 25226 permission unless MAP_PRIVATE is specified in the *flags* parameter as described below.

25227 The parameter *flags* provides other information about the handling of the mapped data. The
 25228 value of *flags* is the bitwise-inclusive OR of these options, defined in <sys/mman.h>:

25229

25230

25231

25232

25233

Symbolic Constant	Description
MAP_SHARED	Changes are shared.
MAP_PRIVATE	Changes are private.
MAP_FIXED	Interpret <i>addr</i> exactly.

25234 Implementations that do not support the Memory Mapped Files option are not required to
 25235 support MAP_PRIVATE.

25236 XSI It is implementation-defined whether MAP_FIXED shall be supported. MAP_FIXED shall be
 25237 supported on XSI-conformant systems.

25238 MAP_SHARED and MAP_PRIVATE describe the disposition of write references to the memory
 25239 object. If MAP_SHARED is specified, write references shall change the underlying object. If
 25240 MAP_PRIVATE is specified, modifications to the mapped data by the calling process shall be
 25241 visible only to the calling process and shall not change the underlying object. It is unspecified
 25242 whether modifications to the underlying object done after the MAP_PRIVATE mapping is
 25243 established are visible through the MAP_PRIVATE mapping. Either MAP_SHARED or
 25244 MAP_PRIVATE can be specified, but not both. The mapping type is retained across *fork*().

25245 TYM When *fildes* represents a typed memory object opened with either the
 25246 POSIX_TYPED_MEM_ALLOCATE flag or the POSIX_TYPED_MEM_ALLOCATE_CONTIG
 25247 flag, *mmap*() shall, if there are enough resources available, map *len* bytes allocated from the
 25248 corresponding typed memory object which were not previously allocated to any process in any
 25249 processor that may access that typed memory object. If there are not enough resources available,
 25250 the function shall fail. If *fildes* represents a typed memory object opened with the
 25251 POSIX_TYPED_MEM_ALLOCATE_CONTIG flag, these allocated bytes shall be contiguous
 25252 within the typed memory object. If *fildes* represents a typed memory object opened with the
 25253 POSIX_TYPED_MEM_ALLOCATE flag, these allocated bytes may be composed of non-
 25254 contiguous fragments within the typed memory object. If *fildes* represents a typed memory
 25255 object opened with neither the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag nor the
 25256 POSIX_TYPED_MEM_ALLOCATE flag, *len* bytes starting at offset *off* within the typed memory
 25257 object are mapped, exactly as when mapping a file or shared memory object. In this case, if two
 25258 processes map an area of typed memory using the same *off* and *len* values and using file
 25259 descriptors that refer to the same memory pool (either from the same port or from a different
 25260 port), both processes shall map the same region of storage.

25261 When MAP_FIXED is set in the *flags* argument, the implementation is informed that the value of
 25262 *pa* shall be *addr*, exactly. If MAP_FIXED is set, *mmap*() may return MAP_FAILED and set *errno* to
 25263 [EINVAL]. If a MAP_FIXED request is successful, the mapping established by *mmap*() replaces
 25264 any previous mappings for the process' pages in the range [*pa*,*pa*+*len*).

When MAP_FIXED is not set, the implementation uses *addr* in an implementation-defined manner to arrive at *pa*. The *pa* so chosen shall be an area of the address space that the implementation deems suitable for a mapping of *len* bytes to the file. All implementations interpret an *addr* value of 0 as granting the implementation complete freedom in selecting *pa*, subject to constraints described below. A non-zero value of *addr* is taken to be a suggestion of a process address near which the mapping should be placed. When the implementation selects a value for *pa*, it never places a mapping at address 0, nor does it replace any extant mapping.

The *off* argument is constrained to be aligned and sized according to the value returned by *sysconf()* when passed *_SC_PAGESIZE* or *_SC_PAGE_SIZE*. When MAP_FIXED is specified, the application shall ensure that the argument *addr* also meets these constraints. The implementation performs mapping operations over whole pages. Thus, while the argument *len* need not meet a size or alignment constraint, the implementation shall include, in any mapping operation, any partial page specified by the range [*pa*,*pa+len*).

The system shall always zero-fill any partial page at the end of an object. Further, the system shall never write out any modified portions of the last page of an object which are beyond its end. References within the address range starting at *pa* and continuing for *len* bytes to whole pages following the end of an object shall result in delivery of a SIGBUS signal.

An implementation may generate SIGBUS signals when a reference would cause an error in the mapped object, such as out-of-space condition.

The *mmap()* function shall add an extra reference to the file associated with the file descriptor *fd* which is not removed by a subsequent *close()* on that file descriptor. This reference shall be removed when there are no more mappings to the file.

The *st_atime* field of the mapped file may be marked for update at any time between the *mmap()* call and the corresponding *munmap()* call. The initial read or write reference to a mapped region shall cause the file's *st_atime* field to be marked for update if it has not already been marked for update.

The *st_ctime* and *st_mtime* fields of a file that is mapped with MAP_SHARED and PROT_WRITE shall be marked for update at some point in the interval between a write reference to the mapped region and the next call to *msync()* with MS_ASYNC or MS_SYNC for that portion of the file by any process. If there is no such call and if the underlying file is modified as a result of a write reference, then these fields shall be marked for update at some time after the write reference.

There may be implementation-defined limits on the number of memory regions that can be mapped (per process or per system).

If such a limit is imposed, whether the number of memory regions that can be mapped by a process is decreased by the use of *shmat()* is implementation-defined.

If *mmap()* fails for reasons other than [EBADF], [EINVAL], or [ENOTSUP], some of the mappings in the address range starting at *addr* and continuing for *len* bytes may have been unmapped.

25304 RETURN VALUE

Upon successful completion, the *mmap()* function shall return the address at which the mapping was placed (*pa*); otherwise, it shall return a value of MAP_FAILED and set *errno* to indicate the error. The symbol MAP_FAILED is defined in the <sys/mman.h> header. No successful return from *mmap()* shall return the value MAP_FAILED.

25309 **ERRORS**

25310		The <i>mmap()</i> function shall fail if:
25311	[EACCES]	The <i>fildest</i> argument is not open for read, regardless of the protection specified, or <i>fildest</i> is not open for write and PROT_WRITE was specified for a MAP_SHARED type mapping.
25312		
25313		
25314 ML	[EAGAIN]	The mapping could not be locked in memory, if required by <i>mlockall()</i> , due to a lack of resources.
25315		
25316	[EBADF]	The <i>fildest</i> argument is not a valid open file descriptor.
25317	[EINVAL]	The <i>addr</i> argument (if MAP_FIXED was specified) or <i>off</i> is not a multiple of the page size as returned by <i>sysconf()</i> , or is considered invalid by the implementation.
25318		
25319		
25320	[EINVAL]	The value of <i>flags</i> is invalid (neither MAP_PRIVATE nor MAP_SHARED is set).
25321		
25322	[EMFILE]	The number of mapped regions would exceed an implementation-defined limit (per process or per system).
25323		
25324	[ENODEV]	The <i>fildest</i> argument refers to a file whose type is not supported by <i>mmap()</i> .
25325	[ENOMEM]	MAP_FIXED was specified, and the range [<i>addr</i> , <i>addr+len</i>) exceeds that allowed for the address space of a process; or, if MAP_FIXED was not specified and there is insufficient room in the address space to effect the mapping.
25326		
25327		
25328 ML	[ENOMEM]	The mapping could not be locked in memory, if required by <i>mlockall()</i> , because it would require more space than the system is able to supply.
25329		
25330 TYM	[ENOMEM]	Not enough unallocated memory resources remain in the typed memory object designated by <i>fildest</i> to allocate <i>len</i> bytes.
25331		
25332	[ENOTSUP]	MAP_FIXED or MAP_PRIVATE was specified in the <i>flags</i> argument and the implementation does not support this functionality.
25333		
25334		The implementation does not support the combination of accesses requested in the <i>prot</i> argument.
25335		
25336	[ENXIO]	Addresses in the range [<i>off</i> , <i>off+len</i>) are invalid for the object specified by <i>fildest</i> .
25337	[ENXIO]	MAP_FIXED was specified in <i>flags</i> and the combination of <i>addr</i> , <i>len</i> , and <i>off</i> is invalid for the object specified by <i>fildest</i> .
25338		
25339 TYM	[ENXIO]	The <i>fildest</i> argument refers to a typed memory object that is not accessible from the calling process.
25340		
25341	[EOVERFLOW]	The file is a regular file and the value of <i>off</i> plus <i>len</i> exceeds the offset maximum established in the open file description associated with <i>fildest</i> .
25342		

25343 **EXAMPLES**

25344 None.

25345 **APPLICATION USAGE**

25346 Use of *mmap()* may reduce the amount of memory available to other memory allocation functions.

25347

25348 Use of MAP_FIXED may result in unspecified behavior in further use of *malloc()* and *shmat()*.

25349 The use of MAP_FIXED is discouraged, as it may prevent an implementation from making the most effective use of resources.

25350

25351 The application must ensure correct synchronization when using *mmap()* in conjunction with
 25352 any other file access method, such as *read()* and *write()*, standard input/output, and *shmat()*.

25353 The *mmap()* function allows access to resources via address space manipulations, instead of
 25354 *read()/write()*. Once a file is mapped, all a process has to do to access it is use the data at the
 25355 address to which the file was mapped. So, using pseudo-code to illustrate the way in which an
 25356 existing program might be changed to use *mmap()*, the following:

```
25357 fildes = open(...)
25358 lseek(fildes, some_offset)
25359 read(fildes, buf, len)
25360 /* Use data in buf. */
```

25361 becomes:

```
25362 fildes = open(...)
25363 address = mmap(0, len, PROT_READ, MAP_PRIVATE, fildes, some_offset)
25364 /* Use data at address. */
```

25365 RATIONALE

25366 After considering several other alternatives, it was decided to adopt the *mmap()* definition found
 25367 in SVR4 for mapping memory objects into process address spaces. The SVR4 definition is
 25368 minimal, in that it describes only what has been built, and what appears to be necessary for a
 25369 general and portable mapping facility.

25370 Note that while *mmap()* was first designed for mapping files, it is actually a general-purpose
 25371 mapping facility. It can be used to map any appropriate object, such as memory, files, devices,
 25372 and so on, into the address space of a process.

25373 When a mapping is established, it is possible that the implementation may need to map more
 25374 than is requested into the address space of the process because of hardware requirements. An
 25375 application, however, cannot count on this behavior. Implementations that do not use a paged
 25376 architecture may simply allocate a common memory region and return the address of it; such
 25377 implementations probably do not allocate any more than is necessary. References past the end of
 25378 the requested area are unspecified.

25379 If an application requests a mapping that would overlay existing mappings in the process, it
 25380 might be desirable that an implementation detect this and inform the application. However, the
 25381 default, portable (not *MAP_FIXED*) operation does not overlay existing mappings. On the other
 25382 hand, if the program specifies a fixed address mapping (which requires some implementation
 25383 knowledge to determine a suitable address, if the function is supported at all), then the program
 25384 is presumed to be successfully managing its own address space and should be trusted when it
 25385 asks to map over existing data structures. Furthermore, it is also desirable to make as few system
 25386 calls as possible, and it might be considered onerous to require an *munmap()* before an *mmap()*
 25387 to the same address range. This volume of IEEE Std 1003.1-2001 specifies that the new mappings
 25388 replace any existing mappings, following existing practice in this regard.

25389 It is not expected, when the Memory Protection option is supported, that all hardware
 25390 implementations are able to support all combinations of permissions at all addresses. When this
 25391 option is supported, implementations are required to disallow write access to mappings without
 25392 write permission and to disallow access to mappings without any access permission. Other than
 25393 these restrictions, implementations may allow access types other than those requested by the
 25394 application. For example, if the application requests only *PROT_WRITE*, the implementation
 25395 may also allow read access. A call to *mmap()* fails if the implementation cannot support allowing
 25396 all the access requested by the application. For example, some implementations cannot support
 25397 a request for both write access and execute access simultaneously. All implementations
 25398 supporting the Memory Protection option must support requests for no access, read access,

write access, and both read and write access. Strictly conforming code must only rely on the required checks. These restrictions allow for portability across a wide range of hardware.

The MAP_FIXED address treatment is likely to fail for non-page-aligned values and for certain architecture-dependent address ranges. Conforming implementations cannot count on being able to choose address values for MAP_FIXED without utilizing non-portable, implementation-defined knowledge. Nonetheless, MAP_FIXED is provided as a standard interface conforming to existing practice for utilizing such knowledge when it is available.

Similarly, in order to allow implementations that do not support virtual addresses, support for directly specifying any mapping addresses via MAP_FIXED is not required and thus a conforming application may not count on it.

The MAP_PRIVATE function can be implemented efficiently when memory protection hardware is available. When such hardware is not available, implementations can implement such “mappings” by simply making a real copy of the relevant data into process private memory, though this tends to behave similarly to *read()*.

The function has been defined to allow for many different models of using shared memory. However, all uses are not equally portable across all machine architectures. In particular, the *mmap()* function allows the system as well as the application to specify the address at which to map a specific region of a memory object. The most portable way to use the function is always to let the system choose the address, specifying NULL as the value for the argument *addr* and not to specify MAP_FIXED.

If it is intended that a particular region of a memory object be mapped at the same address in a group of processes (on machines where this is even possible), then MAP_FIXED can be used to pass in the desired mapping address. The system can still be used to choose the desired address if the first such mapping is made without specifying MAP_FIXED, and then the resulting mapping address can be passed to subsequent processes for them to pass in via MAP_FIXED. The availability of a specific address range cannot be guaranteed, in general.

The *mmap()* function can be used to map a region of memory that is larger than the current size of the object. Memory access within the mapping but beyond the current end of the underlying objects may result in SIGBUS signals being sent to the process. The reason for this is that the size of the object can be manipulated by other processes and can change at any moment. The implementation should tell the application that a memory reference is outside the object where this can be detected; otherwise, written data may be lost and read data may not reflect actual data in the object.

Note that references beyond the end of the object do not extend the object as the new end cannot be determined precisely by most virtual memory hardware. Instead, the size can be directly manipulated by *ftruncate()*.

Process memory locking does apply to shared memory regions, and the MEMLOCK_FUTURE argument to *mlockall()* can be relied upon to cause new shared memory regions to be automatically locked.

Existing implementations of *mmap()* return the value -1 when unsuccessful. Since the casting of this value to type **void *** cannot be guaranteed by the ISO C standard to be distinct from a successful value, this volume of IEEE Std 1003.1-2001 defines the symbol MAP_FAILED, which a conforming implementation does not return as the result of a successful call.

FUTURE DIRECTIONS

None.

25444 **SEE ALSO**

25445 *exec*, *fcntl()*, *fork()*, *lockf()*, *msync()*, *munmap()*, *mprotect()*, *posix_typed_mem_open()*, *shmat()*,
 25446 *sysconf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/mman.h>

25447 **CHANGE HISTORY**

25448 First released in Issue 4, Version 2.

25449 **Issue 5**

25450 Moved from X/OPEN UNIX extension to BASE.

25451 Aligned with *mmap()* in the POSIX Realtime Extension as follows:

- 25452 • The DESCRIPTION is extensively reworded.
- 25453 • The [EAGAIN] and [ENOTSUP] mandatory error conditions are added.
- 25454 • New cases of [ENOMEM] and [ENXIO] are added as mandatory error conditions.
- 25455 • The value returned on failure is the value of the constant MAP_FAILED; this was previously
- 25456 defined as -1.

25457 Large File Summit extensions are added.

25458 **Issue 6**

25459 The *mmap()* function is marked as part of the Memory Mapped Files option.

25460 The Open Group Corrigendum U028/6 is applied, changing (void *)-1 to MAP_FAILED.

25461 The following new requirements on POSIX implementations derive from alignment with the
 25462 Single UNIX Specification:

- 25463 • The DESCRIPTION is updated to described the use of MAP_FIXED.
- 25464 • The DESCRIPTION is updated to describe the addition of an extra reference to the file
- 25465 associated with the file descriptor passed to *mmap()*.
- 25466 • The DESCRIPTION is updated to state that there may be implementation-defined limits on
- 25467 the number of memory regions that can be mapped.
- 25468 • The DESCRIPTION is updated to describe constraints on the alignment and size of the *off*
- 25469 argument.
- 25470 • The [EINVAL] and [EMFILE] error conditions are added.
- 25471 • The [EOVERFLOW] error condition is added. This change is to support large files.

25472 The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- 25473 • The DESCRIPTION is updated to describe the cases when MAP_PRIVATE and MAP_FIXED
- 25474 need not be supported.

25475 The following changes are made for alignment with IEEE Std 1003.1j-2000:

- 25476 • Semantics for typed memory objects are added to the DESCRIPTION.
- 25477 • New [ENOMEM] and [ENXIO] errors are added to the ERRORS section.
- 25478 • The *posix_typed_mem_open()* function is added to the SEE ALSO section.

25479 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

25480 **NAME**

25481 modf, modff, modfl — decompose a floating-point number

25482 **SYNOPSIS**

25483 #include <math.h>

25484 double modf(double *x*, double **iptr*);

25485 float modff(float *value*, float **iptr*);

25486 long double modfl(long double *value*, long double **iptr*);

25487 **DESCRIPTION**

25488 CX The functionality described on this reference page is aligned with the ISO C standard. Any
25489 conflict between the requirements described here and the ISO C standard is unintentional. This
25490 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

25491 These functions shall break the argument *x* into integral and fractional parts, each of which has
25492 the same sign as the argument. It stores the integral part as a **double** (for the *modf*() function), a
25493 **float** (for the *modff*() function), or a **long double** (for the *modfl*() function), in the object pointed
25494 to by *iptr*.

25495 **RETURN VALUE**

25496 Upon successful completion, these functions shall return the signed fractional part of *x*.

25497 MX If *x* is NaN, a NaN shall be returned, and **iptr* shall be set to a NaN.

25498 If *x* is $\pm\text{Inf}$, ± 0 shall be returned, and **iptr* shall be set to $\pm\text{Inf}$.

25499 **ERRORS**

25500 No errors are defined.

25501 **EXAMPLES**

25502 None.

25503 **APPLICATION USAGE**

25504 The *modf*() function computes the function result and **iptr* such that:

25505 a = modf(x, &iptr) ;

25506 x == a+*iptr ;

25507 allowing for the usual floating-point inaccuracies.

25508 **RATIONALE**

25509 None.

25510 **FUTURE DIRECTIONS**

25511 None.

25512 **SEE ALSO**

25513 *frexp*(), *isnan*(), *ldexp*(), the Base Definitions volume of IEEE Std 1003.1-2001, <math.h>

25514 **CHANGE HISTORY**

25515 First released in Issue 1. Derived from Issue 1 of the SVID.

25516 **Issue 5**

25517 The DESCRIPTION is updated to indicate how an application should check for an error. This
25518 text was previously published in the APPLICATION USAGE section.

25519 **Issue 6**

25520 The *modff*() and *modfl*() functions are added for alignment with the ISO/IEC 9899:1999
25521 standard.

25522 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
25523 revised to align with the ISO/IEC 9899:1999 standard.

25524 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
25525 marked.

25526 **NAME**

25527 mprotect — set protection of memory mapping

25528 **SYNOPSIS**

25529 MPR #include <sys/mman.h>

25530 int mprotect(void *addr, size_t len, int prot);

25531

25532 **DESCRIPTION**

25533 The *mprotect()* function shall change the access protections to be that specified by *prot* for those
 25534 whole pages containing any part of the address space of the process starting at address *addr* and
 25535 continuing for *len* bytes. The parameter *prot* determines whether read, write, execute, or some
 25536 combination of accesses are permitted to the data being mapped. The *prot* argument should be
 25537 either PROT_NONE or the bitwise-inclusive OR of one or more of PROT_READ, PROT_WRITE,
 25538 and PROT_EXEC.

25539 If an implementation cannot support the combination of access types specified by *prot*, the call
 25540 to *mprotect()* shall fail.

25541 An implementation may permit accesses other than those specified by *prot*; however, no
 25542 implementation shall permit a write to succeed where PROT_WRITE has not been set or shall
 25543 permit any access where PROT_NONE alone has been set. Implementations shall support at
 25544 least the following values of *prot*: PROT_NONE, PROT_READ, PROT_WRITE, and the bitwise-
 25545 inclusive OR of PROT_READ and PROT_WRITE. If PROT_WRITE is specified, the application
 25546 shall ensure that it has opened the mapped objects in the specified address range with write
 25547 permission, unless MAP_PRIVATE was specified in the original mapping, regardless of whether
 25548 the file descriptors used to map the objects have since been closed.

25549 The implementation shall require that *addr* be a multiple of the page size as returned by
 25550 *sysconf()*.

25551 The behavior of this function is unspecified if the mapping was not established by a call to
 25552 *mmap()*.

25553 When *mprotect()* fails for reasons other than [EINVAL], the protections on some of the pages in
 25554 the range [*addr*,*addr*+*len*) may have been changed.

25555 **RETURN VALUE**

25556 Upon successful completion, *mprotect()* shall return 0; otherwise, it shall return -1 and set *errno*
 25557 to indicate the error.

25558 **ERRORS**25559 The *mprotect()* function shall fail if:

25560 [EACCES] The *prot* argument specifies a protection that violates the access permission
 25561 the process has to the underlying memory object.

25562 [EAGAIN] The *prot* argument specifies PROT_WRITE over a MAP_PRIVATE mapping
 25563 and there are insufficient memory resources to reserve for locking the private
 25564 page.

25565 [EINVAL] The *addr* argument is not a multiple of the page size as returned by *sysconf()*.

25566 [ENOMEM] Addresses in the range [*addr*,*addr*+*len*) are invalid for the address space of a
 25567 process, or specify one or more pages which are not mapped.

25568 [ENOMEM] The *prot* argument specifies PROT_WRITE on a MAP_PRIVATE mapping, and
 25569 it would require more space than the system is able to supply for locking the
 25570 private pages, if required.

25571 [ENOTSUP] The implementation does not support the combination of accesses requested
 25572 in the *prot* argument.

25573 EXAMPLES

25574 None.

25575 APPLICATION USAGE

25576 The [EINVAL] error above is marked EX because it is defined as an optional error in the POSIX
 25577 Realtime Extension.

25578 RATIONALE

25579 None.

25580 FUTURE DIRECTIONS

25581 None.

25582 SEE ALSO

25583 *mmap()*, *sysconf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/mman.h>

25584 CHANGE HISTORY

25585 First released in Issue 4, Version 2.

25586 Issue 5

25587 Moved from X/OPEN UNIX extension to BASE.

25588 Aligned with *mprotect()* in the POSIX Realtime Extension as follows:

- 25589 • The DESCRIPTION is largely reworded.
- 25590 • [ENOTSUP] and a second form of [ENOMEM] are added as mandatory error conditions.
- 25591 • [EAGAIN] is moved from the optional to the mandatory error conditions.

25592 Issue 6

25593 The *mprotect()* function is marked as part of the Memory Protection option.

25594 The following new requirements on POSIX implementations derive from alignment with the
 25595 Single UNIX Specification:

- 25596 • The DESCRIPTION is updated to state that implementations require *addr* to be a multiple of
 25597 the page size as returned by *sysconf()*.
- 25598 • The [EINVAL] error condition is added.

25599 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

25600 **NAME**25601 mq_close — close a message queue (**REALTIME**)25602 **SYNOPSIS**

25603 MSG #include <mqueue.h>

25604 int mq_close(mqd_t *mqdes*);

25605

25606 **DESCRIPTION**

25607 The *mq_close()* function shall remove the association between the message queue descriptor, *mqdes*, and its message queue. The results of using this message queue descriptor after successful return from this *mq_close()*, and until the return of this message queue descriptor from a subsequent *mq_open()*, are undefined.

25611 If the process has successfully attached a notification request to the message queue via this *mqdes*, this attachment shall be removed, and the message queue is available for another process to attach for notification.

25614 **RETURN VALUE**

25615 Upon successful completion, the *mq_close()* function shall return a value of zero; otherwise, the function shall return a value of -1 and set *errno* to indicate the error.

25617 **ERRORS**25618 The *mq_close()* function shall fail if:

25619 [EBADF] The *mqdes* argument is not a valid message queue descriptor.

25620 **EXAMPLES**

25621 None.

25622 **APPLICATION USAGE**

25623 None.

25624 **RATIONALE**

25625 None.

25626 **FUTURE DIRECTIONS**

25627 None.

25628 **SEE ALSO**

25629 *mq_open()*, *mq_unlink()*, *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*, the Base Definitions volume of IEEE Std 1003.1-2001, <mqueue.h>

25631 **CHANGE HISTORY**

25632 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

25633 **Issue 6**25634 The *mq_close()* function is marked as part of the Message Passing option.

25635 The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.

25637 **NAME**25638 mq_getattr — get message queue attributes (**REALTIME**)25639 **SYNOPSIS**

25640 MSG #include <mqueue.h>

25641 int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat);

25642

25643 **DESCRIPTION**25644 The *mq_getattr()* function shall obtain status information and attributes of the message queue
25645 and the open message queue description associated with the message queue descriptor.25646 The *mqdes* argument specifies a message queue descriptor.25647 The results shall be returned in the **mq_attr** structure referenced by the *mqstat* argument.25648 Upon return, the following members shall have the values associated with the open message
25649 queue description as set when the message queue was opened and as modified by subsequent
25650 *mq_setattr()* calls: *mq_flags*.25651 The following attributes of the message queue shall be returned as set at message queue
25652 creation: *mq_maxmsg*, *mq_msgsize*.25653 Upon return, the following members within the **mq_attr** structure referenced by the *mqstat*
25654 argument shall be set to the current state of the message queue:25655 *mq_curmsgs* The number of messages currently on the queue.25656 **RETURN VALUE**25657 Upon successful completion, the *mq_getattr()* function shall return zero. Otherwise, the function
25658 shall return -1 and set *errno* to indicate the error.25659 **ERRORS**25660 The *mq_getattr()* function shall fail if:25661 [EBADF] The *mqdes* argument is not a valid message queue descriptor.25662 **EXAMPLES**

25663 None.

25664 **APPLICATION USAGE**

25665 None.

25666 **RATIONALE**

25667 None.

25668 **FUTURE DIRECTIONS**

25669 None.

25670 **SEE ALSO**25671 *mq_open()*, *mq_send()*, *mq_setattr()*, *mq_timedsend()*, *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*, the
25672 Base Definitions volume of IEEE Std 1003.1-2001, <mqueue.h>25673 **CHANGE HISTORY**

25674 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

25675 **Issue 6**25676 The *mq_getattr()* function is marked as part of the Message Passing option.25677 The [ENOSYS] error condition has been removed as stubs need not be provided if an
25678 implementation does not support the Message Passing option.

25679 The *mq_timedsend()* function is added to the SEE ALSO section for alignment with
25680 IEEE Std 1003.1d-1999.

25681 **NAME**25682 mq_notify — notify process that a message is available (**REALTIME**)25683 **SYNOPSIS**

25684 MSG #include <mqueue.h>

25685 int mq_notify(mqd_t mqdes, const struct sigevent *notification);

25686

25687 **DESCRIPTION**

25688 If the argument *notification* is not NULL, this function shall register the calling process to be
 25689 notified of message arrival at an empty message queue associated with the specified message
 25690 queue descriptor, *mqdes*. The notification specified by the *notification* argument shall be sent to
 25691 the process when the message queue transitions from empty to non-empty. At any time, only
 25692 one process may be registered for notification by a message queue. If the calling process or any
 25693 other process has already registered for notification of message arrival at the specified message
 25694 queue, subsequent attempts to register for that message queue shall fail.

25695 If *notification* is NULL and the process is currently registered for notification by the specified
 25696 message queue, the existing registration shall be removed.

25697 When the notification is sent to the registered process, its registration shall be removed. The
 25698 message queue shall then be available for registration.

25699 If a process has registered for notification of message arrival at a message queue and some
 25700 thread is blocked in *mq_receive()* waiting to receive a message when a message arrives at the
 25701 queue, the arriving message shall satisfy the appropriate *mq_receive()*. The resulting behavior is
 25702 as if the message queue remains empty, and no notification shall be sent.

25703 **RETURN VALUE**

25704 Upon successful completion, the *mq_notify()* function shall return a value of zero; otherwise, the
 25705 function shall return a value of -1 and set *errno* to indicate the error.

25706 **ERRORS**25707 The *mq_notify()* function shall fail if:25708 [EBADF] The *mqdes* argument is not a valid message queue descriptor.

25709 [EBUSY] A process is already registered for notification by the message queue.

25710 **EXAMPLES**

25711 None.

25712 **APPLICATION USAGE**

25713 None.

25714 **RATIONALE**

25715 None.

25716 **FUTURE DIRECTIONS**

25717 None.

25718 **SEE ALSO**

25719 *mq_open()*, *mq_send()*, *mq_timedsend()*, *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*, the Base
 25720 Definitions volume of IEEE Std 1003.1-2001, <mqueue.h>

25721 **CHANGE HISTORY**

25722 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

25723 Issue 6

25724 The *mq_notify()* function is marked as part of the Message Passing option.

25725 The [ENOSYS] error condition has been removed as stubs need not be provided if an
25726 implementation does not support the Message Passing option.

25727 The *mq_timedsend()* function is added to the SEE ALSO section for alignment with
25728 IEEE Std 1003.1d-1999.

25729 **NAME**25730 `mq_open` — open a message queue (**REALTIME**)25731 **SYNOPSIS**25732 MSG `#include <mqueue.h>`25733 `mqd_t mq_open(const char *name, int oflag, ...);`

25734

25735 **DESCRIPTION**

25736 The `mq_open()` function shall establish the connection between a process and a message queue
 25737 with a message queue descriptor. It shall create an open message queue description that refers to
 25738 the message queue, and a message queue descriptor that refers to that open message queue
 25739 description. The message queue descriptor is used by other functions to refer to that message
 25740 queue. The *name* argument points to a string naming a message queue. It is unspecified whether
 25741 the name appears in the file system and is visible to other functions that take pathnames as
 25742 arguments. The *name* argument shall conform to the construction rules for a pathname. If *name*
 25743 begins with the slash character, then processes calling `mq_open()` with the same value of *name*
 25744 shall refer to the same message queue object, as long as that name has not been removed. If *name*
 25745 does not begin with the slash character, the effect is implementation-defined. The interpretation
 25746 of slash characters other than the leading slash character in *name* is implementation-defined. If
 25747 the *name* argument is not the name of an existing message queue and creation is not requested,
 25748 `mq_open()` shall fail and return an error.

25749 A message queue descriptor may be implemented using a file descriptor, in which case
 25750 applications can open up to at least {OPEN_MAX} file and message queues.

25751 The *oflag* argument requests the desired receive and/or send access to the message queue. The
 25752 requested access permission to receive messages or send messages shall be granted if the calling
 25753 process would be granted read or write access, respectively, to an equivalently protected file.

25754 The value of *oflag* is the bitwise-inclusive OR of values from the following list. Applications
 25755 shall specify exactly one of the first three values (access modes) below in the value of *oflag*:

25756 **O_RDONLY** Open the message queue for receiving messages. The process can use the
 25757 returned message queue descriptor with `mq_receive()`, but not `mq_send()`. A
 25758 message queue may be open multiple times in the same or different processes
 25759 for receiving messages.

25760 **O_WRONLY** Open the queue for sending messages. The process can use the returned
 25761 message queue descriptor with `mq_send()` but not `mq_receive()`. A message
 25762 queue may be open multiple times in the same or different processes for
 25763 sending messages.

25764 **O_RDWR** Open the queue for both receiving and sending messages. The process can use
 25765 any of the functions allowed for **O_RDONLY** and **O_WRONLY**. A message
 25766 queue may be open multiple times in the same or different processes for
 25767 sending messages.

25768 Any combination of the remaining flags may be specified in the value of *oflag*:

25769 **O_CREAT** Create a message queue. It requires two additional arguments: *mode*, which
 25770 shall be of type **mode_t**, and *attr*, which shall be a pointer to an **mq_attr**
 25771 structure. If the pathname *name* has already been used to create a message
 25772 queue that still exists, then this flag shall have no effect, except as noted under
 25773 **O_EXCL**. Otherwise, a message queue shall be created without any messages
 25774 in it. The user ID of the message queue shall be set to the effective user ID of
 25775 the process, and the group ID of the message queue shall be set to the effective

25776		group ID of the process. The file permission bits shall be set to the value of <i>mode</i> . When bits in <i>mode</i> other than file permission bits are set, the effect is implementation-defined. If <i>attr</i> is NULL, the message queue shall be created with implementation-defined default message queue attributes. If <i>attr</i> is non-NULL and the calling process has the appropriate privilege on <i>name</i> , the message queue <i>mq_maxmsg</i> and <i>mq_msgsize</i> attributes shall be set to the values of the corresponding members in the mq_attr structure referred to by <i>attr</i> . If <i>attr</i> is non-NULL, but the calling process does not have the appropriate privilege on <i>name</i> , the <i>mq_open()</i> function shall fail and return an error without creating the message queue.
25786	O_EXCL	If O_EXCL and O_CREAT are set, <i>mq_open()</i> shall fail if the message queue <i>name</i> exists. The check for the existence of the message queue and the creation of the message queue if it does not exist shall be atomic with respect to other threads executing <i>mq_open()</i> naming the same <i>name</i> with O_EXCL and O_CREAT set. If O_EXCL is set and O_CREAT is not set, the result is undefined.
25792	O_NONBLOCK	Determines whether an <i>mq_send()</i> or <i>mq_receive()</i> waits for resources or messages that are not currently available, or fails with <i>errno</i> set to [EAGAIN]; see <i>mq_send()</i> and <i>mq_receive()</i> for details.
25795	The <i>mq_open()</i> function does not add or remove messages from the queue.	
25796	RETURN VALUE	
25797	Upon successful completion, the function shall return a message queue descriptor; otherwise, the function shall return (mqd_t)−1 and set <i>errno</i> to indicate the error.	
25799	ERRORS	
25800	The <i>mq_open()</i> function shall fail if:	
25801	[EACCES]	The message queue exists and the permissions specified by <i>oflag</i> are denied, or the message queue does not exist and permission to create the message queue is denied.
25804	[EEXIST]	O_CREAT and O_EXCL are set and the named message queue already exists.
25805	[EINTR]	The <i>mq_open()</i> function was interrupted by a signal.
25806	[EINVAL]	The <i>mq_open()</i> function is not supported for the given name.
25807	[EINVAL]	O_CREAT was specified in <i>oflag</i> , the value of <i>attr</i> is not NULL, and either <i>mq_maxmsg</i> or <i>mq_msgsize</i> was less than or equal to zero.
25809	[EMFILE]	Too many message queue descriptors or file descriptors are currently in use by this process.
25811	[ENAMETOOLONG]	The length of the <i>name</i> argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
25814	[ENFILE]	Too many message queues are currently open in the system.
25815	[ENOENT]	O_CREAT is not set and the named message queue does not exist.
25816	[ENOSPC]	There is insufficient space for the creation of the new message queue.

25817 **EXAMPLES**

25818 None.

25819 **APPLICATION USAGE**

25820 None.

25821 **RATIONALE**

25822 None.

25823 **FUTURE DIRECTIONS**

25824 None.

25825 **SEE ALSO**

25826 *mq_close()*, *mq_getattr()*, *mq_receive()*, *mq_send()*, *mq_setattr()*, *mq_timedreceive()*, *mq_timedsend()*,
25827 *mq_unlink()*, *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*, the Base Definitions volume of
25828 IEEE Std 1003.1-2001, <mqqueue.h>

25829 **CHANGE HISTORY**

25830 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

25831 **Issue 6**25832 The *mq_open()* function is marked as part of the Message Passing option.

25833 The [ENOSYS] error condition has been removed as stubs need not be provided if an
25834 implementation does not support the Message Passing option.

25835 The *mq_timedreceive()* and *mq_timedsend()* functions are added to the SEE ALSO section for
25836 alignment with IEEE Std 1003.1d-1999.

25837 The DESCRIPTION of O_EXCL is updated in response to IEEE PASC Interpretation 1003.1c #48.

25838 NAME

25839 `mq_receive`, `mq_timedreceive` — receive a message from a message queue (**REALTIME**)

25840 SYNOPSIS

25841 MSG `#include <mqueue.h>`

25842 `ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,`
 25843 `unsigned *msg_prio);`

25844

25845 MSG TMO `#include <mqueue.h>`

25846 `#include <time.h>`

25847 `ssize_t mq_timedreceive(mqd_t mqdes, char *restrict msg_ptr,`
 25848 `size_t msg_len, unsigned *restrict msg_prio,`
 25849 `const struct timespec *restrict abs_timeout);`

25850

25851 DESCRIPTION

25852 The `mq_receive()` function shall receive the oldest of the highest priority message(s) from the
 25853 message queue specified by `mqdes`. If the size of the buffer in bytes, specified by the `msg_len`
 25854 argument, is less than the `mq_msgsize` attribute of the message queue, the function shall fail and
 25855 return an error. Otherwise, the selected message shall be removed from the queue and copied to
 25856 the buffer pointed to by the `msg_ptr` argument.

25857 If the value of `msg_len` is greater than {SSIZE_MAX}, the result is implementation-defined.

25858 If the argument `msg_prio` is not NULL, the priority of the selected message shall be stored in the
 25859 location referenced by `msg_prio`.

25860 If the specified message queue is empty and O_NONBLOCK is not set in the message queue
 25861 description associated with `mqdes`, `mq_receive()` shall block until a message is enqueued on the
 25862 message queue or until `mq_receive()` is interrupted by a signal. If more than one thread is waiting
 25863 to receive a message when a message arrives at an empty queue and the Priority Scheduling
 25864 option is supported, then the thread of highest priority that has been waiting the longest shall be
 25865 selected to receive the message. Otherwise, it is unspecified which waiting thread receives the
 25866 message. If the specified message queue is empty and O_NONBLOCK is set in the message
 25867 queue description associated with `mqdes`, no message shall be removed from the queue, and
 25868 `mq_receive()` shall return an error.

25869 TMO The `mq_timedreceive()` function shall receive the oldest of the highest priority messages from the
 25870 message queue specified by `mqdes` as described for the `mq_receive()` function. However, if
 25871 O_NONBLOCK was not specified when the message queue was opened via the `mq_open()`
 25872 function, and no message exists on the queue to satisfy the receive, the wait for such a message
 25873 shall be terminated when the specified timeout expires. If O_NONBLOCK is set, this function is
 25874 equivalent to `mq_receive()`.

25875 The timeout expires when the absolute time specified by `abs_timeout` passes, as measured by the
 25876 clock on which timeouts are based (that is, when the value of that clock equals or exceeds
 25877 `abs_timeout`), or if the absolute time specified by `abs_timeout` has already been passed at the time
 25878 of the call.

25879 TMO TMR If the Timers option is supported, the timeout shall be based on the CLOCK_REALTIME clock; if
 25880 the Timers option is not supported, the timeout shall be based on the system clock as returned
 25881 by the `time()` function.

25882 TMO The resolution of the timeout shall be the resolution of the clock on which it is based. The
 25883 `timespec` argument is defined in the `<time.h>` header.

25884 Under no circumstance shall the operation fail with a timeout if a message can be removed from
 25885 the message queue immediately. The validity of the *abs_timeout* parameter need not be checked
 25886 if a message can be removed from the message queue immediately.

25887 RETURN VALUE

25888 TMO Upon successful completion, the *mq_receive()* and *mq_timedreceive()* functions shall return the
 25889 length of the selected message in bytes and the message shall be removed from the queue.
 25890 Otherwise, no message shall be removed from the queue, the functions shall return a value of -1,
 25891 and set *errno* to indicate the error.

25892 ERRORS

25893 TMO The *mq_receive()* and *mq_timedreceive()* functions shall fail if:

25894 [EAGAIN] O_NONBLOCK was set in the message description associated with *mqdes*,
 25895 and the specified message queue is empty.

25896 [EBADF] The *mqdes* argument is not a valid message queue descriptor open for reading.

25897 [EMSGSIZE] The specified message buffer size, *msg_len*, is less than the message size
 25898 attribute of the message queue.

25899 TMO [EINTR] The *mq_receive()* or *mq_timedreceive()* operation was interrupted by a signal.

25900 TMO [EINVAL] The process or thread would have blocked, and the *abs_timeout* parameter
 25901 specified a nanoseconds field value less than zero or greater than or equal to
 25902 1 000 million.

25903 TMO [ETIMEDOUT] The O_NONBLOCK flag was not set when the message queue was opened,
 25904 but no message arrived on the queue before the specified timeout expired.

25905 TMO The *mq_receive()* and *mq_timedreceive()* functions may fail if:

25906 [EBADMSG] The implementation has detected a data corruption problem with the
 25907 message.

25908 EXAMPLES

25909 None.

25910 APPLICATION USAGE

25911 None.

25912 RATIONALE

25913 None.

25914 FUTURE DIRECTIONS

25915 None.

25916 SEE ALSO

25917 *mq_open()*, *mq_send()*, *mq_timedsend()*, *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*, *time()*, the Base
 25918 Definitions volume of IEEE Std 1003.1-2001, <mqqueue.h>, <time.h>

25919 CHANGE HISTORY

25920 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

25921 Issue 6

25922 The *mq_receive()* function is marked as part of the Message Passing option.

25923 The Open Group Corrigendum U021/4 is applied. The DESCRIPTION is changed to refer to
 25924 *msg_len* rather than *maxsize*.

25925 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 25926 implementation does not support the Message Passing option.

- 25927 The following new requirements on POSIX implementations derive from alignment with the
25928 Single UNIX Specification:
- 25929 • In this function it is possible for the return value to exceed the range of the type **ssize_t** (since
25930 **size_t** has a larger range of positive values than **ssize_t**). A sentence restricting the size of
25931 the **size_t** object is added to the description to resolve this conflict.
- 25932 The *mq_timedreceive()* function is added for alignment with IEEE Std 1003.1d-1999.
- 25933 The **restrict** keyword is added to the *mq_timedreceive()* prototype for alignment with the
25934 ISO/IEC 9899:1999 standard.
- 25935 IEEE PASC Interpretation 1003.1 #109 is applied, correcting the return type for *mq_timedreceive()*
25936 from **int** to **ssize_t**.

25937 **NAME**25938 `mq_send`, `mq_timedsend` — send a message to a message queue (**REALTIME**)25939 **SYNOPSIS**25940 MSG `#include <mqueue.h>`25941 `int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,`
25942 `unsigned msg_prio);`

25943

25944 MSG TMO `#include <mqueue.h>`25945 `#include <time.h>`25946 `int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len,`
25947 `unsigned msg_prio, const struct timespec *abs_timeout);`

25948

25949 **DESCRIPTION**25950 The `mq_send()` function shall add the message pointed to by the argument `msg_ptr` to the
25951 message queue specified by `mqdes`. The `msg_len` argument specifies the length of the message, in
25952 bytes, pointed to by `msg_ptr`. The value of `msg_len` shall be less than or equal to the `mq_msgsize`
25953 attribute of the message queue, or `mq_send()` shall fail.25954 If the specified message queue is not full, `mq_send()` shall behave as if the message is inserted
25955 into the message queue at the position indicated by the `msg_prio` argument. A message with a
25956 larger numeric value of `msg_prio` shall be inserted before messages with lower values of
25957 `msg_prio`. A message shall be inserted after other messages in the queue, if any, with equal
25958 `msg_prio`. The value of `msg_prio` shall be less than {MQ_PRIO_MAX}.25959 If the specified message queue is full and O_NONBLOCK is not set in the message queue
25960 description associated with `mqdes`, `mq_send()` shall block until space becomes available to
25961 enqueue the message, or until `mq_send()` is interrupted by a signal. If more than one thread is
25962 waiting to send when space becomes available in the message queue and the Priority Scheduling
25963 option is supported, then the thread of the highest priority that has been waiting the longest
25964 shall be unblocked to send its message. Otherwise, it is unspecified which waiting thread is
25965 unblocked. If the specified message queue is full and O_NONBLOCK is set in the message
25966 queue description associated with `mqdes`, the message shall not be queued and `mq_send()` shall
25967 return an error.25968 TMO The `mq_timedsend()` function shall add a message to the message queue specified by `mqdes` in the
25969 manner defined for the `mq_send()` function. However, if the specified message queue is full and
25970 O_NONBLOCK is not set in the message queue description associated with `mqdes`, the wait for
25971 sufficient room in the queue shall be terminated when the specified timeout expires. If
25972 O_NONBLOCK is set in the message queue description, this function shall be equivalent to
25973 `mq_send()`.25974 The timeout shall expire when the absolute time specified by `abs_timeout` passes, as measured by
25975 the clock on which timeouts are based (that is, when the value of that clock equals or exceeds
25976 `abs_timeout`), or if the absolute time specified by `abs_timeout` has already been passed at the time
25977 of the call.25978 TMO TMR If the Timers option is supported, the timeout shall be based on the CLOCK_REALTIME clock; if
25979 the Timers option is not supported, the timeout shall be based on the system clock as returned
25980 by the `time()` function.25981 TMO The resolution of the timeout shall be the resolution of the clock on which it is based. The
25982 `timespec` argument is defined in the `<time.h>` header.

25983 Under no circumstance shall the operation fail with a timeout if there is sufficient room in the
 25984 queue to add the message immediately. The validity of the *abs_timeout* parameter need not be
 25985 checked when there is sufficient room in the queue.

25986 RETURN VALUE

25987 TMO Upon successful completion, the *mq_send()* and *mq_timedsend()* functions shall return a value of
 25988 zero. Otherwise, no message shall be enqueued, the functions shall return `-1`, and *errno* shall be
 25989 set to indicate the error.

25990 ERRORS

25991 TMO The *mq_send()* and *mq_timedsend()* functions shall fail if:

25992 [EAGAIN] The `O_NONBLOCK` flag is set in the message queue description associated
 25993 with *mqdes*, and the specified message queue is full.

25994 [EBADF] The *mqdes* argument is not a valid message queue descriptor open for writing.

25995 TMO [EINTR] A signal interrupted the call to *mq_send()* or *mq_timedsend()*.

25996 [EINVAL] The value of *msg_prio* was outside the valid range.

25997 TMO [EINVAL] The process or thread would have blocked, and the *abs_timeout* parameter
 25998 specified a nanoseconds field value less than zero or greater than or equal to
 25999 1 000 million.

26000 [EMSGSIZE] The specified message length, *msg_len*, exceeds the message size attribute of
 26001 the message queue.

26002 TMO [ETIMEDOUT] The `O_NONBLOCK` flag was not set when the message queue was opened,
 26003 but the timeout expired before the message could be added to the queue.

26004 EXAMPLES

26005 None.

26006 APPLICATION USAGE

26007 The value of the symbol `{MQ_PRIO_MAX}` limits the number of priority levels supported by the
 26008 application. Message priorities range from 0 to `{MQ_PRIO_MAX}-1`.

26009 RATIONALE

26010 None.

26011 FUTURE DIRECTIONS

26012 None.

26013 SEE ALSO

26014 *mq_open()*, *mq_receive()*, *mq_setattr()*, *mq_timedreceive()*, *time()*, the Base Definitions volume of
 26015 IEEE Std 1003.1-2001, `<mqqueue.h>`, `<time.h>`

26016 CHANGE HISTORY

26017 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

26018 Issue 6

26019 The *mq_send()* function is marked as part of the Message Passing option.

26020 The `[ENOSYS]` error condition has been removed as stubs need not be provided if an
 26021 implementation does not support the Message Passing option.

26022 The *mq_timedsend()* function is added for alignment with IEEE Std 1003.1d-1999.

26023 **NAME**26024 mq_setattr — set message queue attributes (**REALTIME**)26025 **SYNOPSIS**

26026 MSG #include <mqueue.h>

```
26027     int mq_setattr(mqd_t mqdes, const struct mq_attr *restrict mqstat,
26028                    struct mq_attr *restrict omqstat);
```

26029

26030 **DESCRIPTION**

26031 The *mq_setattr()* function shall set attributes associated with the open message queue
 26032 description referenced by the message queue descriptor specified by *mqdes*.

26033 The message queue attributes corresponding to the following members defined in the **mq_attr**
 26034 structure shall be set to the specified values upon successful completion of *mq_setattr()*:

26035 *mq_flags* The value of this member is the bitwise-logical OR of zero or more of
 26036 O_NONBLOCK and any implementation-defined flags.

26037 The values of the *mq_maxmsg*, *mq_msgsize*, and *mq_curmsgs* members of the **mq_attr** structure
 26038 shall be ignored by *mq_setattr()*.

26039 If *omqstat* is non-NULL, the *mq_setattr()* function shall store, in the location referenced by
 26040 *omqstat*, the previous message queue attributes and the current queue status. These values shall
 26041 be the same as would be returned by a call to *mq_getattr()* at that point.

26042 **RETURN VALUE**

26043 Upon successful completion, the function shall return a value of zero and the attributes of the
 26044 message queue shall have been changed as specified.

26045 Otherwise, the message queue attributes shall be unchanged, and the function shall return a
 26046 value of -1 and set *errno* to indicate the error.

26047 **ERRORS**

26048 The *mq_setattr()* function shall fail if:

26049 [EBADF] The *mqdes* argument is not a valid message queue descriptor.

26050 **EXAMPLES**

26051 None.

26052 **APPLICATION USAGE**

26053 None.

26054 **RATIONALE**

26055 None.

26056 **FUTURE DIRECTIONS**

26057 None.

26058 **SEE ALSO**

26059 *mq_open()*, *mq_send()*, *mq_timedsend()*, *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*, the Base
 26060 Definitions volume of IEEE Std 1003.1-2001, <mqueue.h>

26061 **CHANGE HISTORY**

26062 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

- 26064 The *mq_setattr()* function is marked as part of the Message Passing option.
- 26065 The [ENOSYS] error condition has been removed as stubs need not be provided if an
26066 implementation does not support the Message Passing option.
- 26067 The *mq_timedsend()* function is added to the SEE ALSO section for alignment with
26068 IEEE Std 1003.1d-1999.
- 26069 The **restrict** keyword is added to the *mq_setattr()* prototype for alignment with the
26070 ISO/IEC 9899:1999 standard.

26071 **NAME**26072 mq_timedreceive — receive a message from a message queue (**ADVANCED REALTIME**)26073 **SYNOPSIS**

26074 MSG TMO #include <mqueue.h>

26075 #include <time.h>

26076 ssize_t mq_timedreceive(mqd_t mqdes, char *restrict msg_ptr,

26077 size_t msg_len, unsigned *restrict msg_prio,

26078 const struct timespec *restrict abs_timeout);

26079

26080 **DESCRIPTION**26081 Refer to *mq_receive()*.

26082 NAME

26083 mq_timedsend — send a message to a message queue (**ADVANCED REALTIME**)

26084 SYNOPSIS

26085 MSG TMO `#include <mqueue.h>`

26086 `#include <time.h>`

```
26087       int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len,  
26088                       unsigned msg_prio, const struct timespec *abs_timeout);
```

26089

26090 DESCRIPTION

26091 Refer to *mq_send()*.

26092 **NAME**26093 mq_unlink — remove a message queue (**REALTIME**)26094 **SYNOPSIS**

26095 MSG #include <mqueue.h>

26096 int mq_unlink(const char *name);

26097

26098 **DESCRIPTION**

26099 The *mq_unlink()* function shall remove the message queue named by the pathname *name*. After
 26100 a successful call to *mq_unlink()* with *name*, a call to *mq_open()* with *name* shall fail if the flag
 26101 O_CREAT is not set in *flags*. If one or more processes have the message queue open when
 26102 *mq_unlink()* is called, destruction of the message queue shall be postponed until all references to
 26103 the message queue have been closed.

26104 Calls to *mq_open()* to recreate the message queue may fail until the message queue is actually
 26105 removed. However, the *mq_unlink()* call need not block until all references have been closed; it
 26106 may return immediately.

26107 **RETURN VALUE**

26108 Upon successful completion, the function shall return a value of zero. Otherwise, the named
 26109 message queue shall be unchanged by this function call, and the function shall return a value of
 26110 −1 and set *errno* to indicate the error.

26111 **ERRORS**26112 The *mq_unlink()* function shall fail if:

26113 [EACCES] Permission is denied to unlink the named message queue.

26114 [ENAMETOOLONG]

26115 The length of the *name* argument exceeds {PATH_MAX} or a pathname
 26116 component is longer than {NAME_MAX}.

26117 [ENOENT] The named message queue does not exist.

26118 **EXAMPLES**

26119 None.

26120 **APPLICATION USAGE**

26121 None.

26122 **RATIONALE**

26123 None.

26124 **FUTURE DIRECTIONS**

26125 None.

26126 **SEE ALSO**

26127 *mq_close()*, *mq_open()*, *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*, the Base Definitions volume of
 26128 IEEE Std 1003.1-2001, <mqueue.h>

26129 **CHANGE HISTORY**

26130 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

26131 **Issue 6**26132 The *mq_unlink()* function is marked as part of the Message Passing option.

26133 The Open Group Corrigendum U021/5 is applied, clarifying that upon unsuccessful completion,
 26134 the named message queue is unchanged by this function.

26135 The [ENOSYS] error condition has been removed as stubs need not be provided if an
26136 implementation does not support the Message Passing option.

26137 **NAME**

26138 mrnd48 — generate uniformly distributed pseudo-random signed long integers

26139 **SYNOPSIS**

26140 xSI #include <stdlib.h>

26141 long mrnd48(void);

26142

26143 **DESCRIPTION**

26144 Refer to *drand48()*.

26145 **NAME**

26146 msgctl — XSI message control operations

26147 **SYNOPSIS**26148 XSI

```
#include <sys/msg.h>
```

26149

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

26150

26151 **DESCRIPTION**

26152 The *msgctl()* function operates on XSI message queues (see the Base Definitions volume of
 26153 IEEE Std 1003.1-2001, Section 3.224, Message Queue). It is unspecified whether this function
 26154 interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on
 26155 page 41).

26156 The *msgctl()* function shall provide message control operations as specified by *cmd*. The
 26157 following values for *cmd*, and the message control operations they specify, are:

26158 **IPC_STAT** Place the current value of each member of the **msqid_ds** data structure
 26159 associated with *msqid* into the structure pointed to by *buf*. The contents of this
 26160 structure are defined in **<sys/msg.h>**.

26161 **IPC_SET** Set the value of the following members of the **msqid_ds** data structure
 26162 associated with *msqid* to the corresponding value found in the structure
 26163 pointed to by *buf*:

26164 msg_perm.uid
 26165 msg_perm.gid
 26166 msg_perm.mode
 26167 msg_qbytes

26168 **IPC_SET** can only be executed by a process with appropriate privileges or that
 26169 has an effective user ID equal to the value of **msg_perm.cuid** or
 26170 **msg_perm.uid** in the **msqid_ds** data structure associated with *msqid*. Only a
 26171 process with appropriate privileges can raise the value of **msg_qbytes**.

26172 **IPC_RMID** Remove the message queue identifier specified by *msqid* from the system and
 26173 destroy the message queue and **msqid_ds** data structure associated with it.
 26174 **IPC_RMID** can only be executed by a process with appropriate privileges or
 26175 one that has an effective user ID equal to the value of **msg_perm.cuid** or
 26176 **msg_perm.uid** in the **msqid_ds** data structure associated with *msqid*.

26177 **RETURN VALUE**

26178 Upon successful completion, *msgctl()* shall return 0; otherwise, it shall return -1 and set *errno* to
 26179 indicate the error.

26180 **ERRORS**26181 The *msgctl()* function shall fail if:

26182 **[EACCES]** The argument *cmd* is **IPC_STAT** and the calling process does not have read
 26183 permission; see Section 2.7 (on page 39).

26184 **[EINVAL]** The value of *msqid* is not a valid message queue identifier; or the value of *cmd*
 26185 is not a valid command.

26186 **[EPERM]** The argument *cmd* is **IPC_RMID** or **IPC_SET** and the effective user ID of the
 26187 calling process is not equal to that of a process with appropriate privileges
 26188 and it is not equal to the value of **msg_perm.cuid** or **msg_perm.uid** in the data
 26189 structure associated with *msqid*.

26190 [EPERM] The argument *cmd* is IPC_SET, an attempt is being made to increase to the
26191 value of **msg_qbytes**, and the effective user ID of the calling process does not
26192 have appropriate privileges.

26193 EXAMPLES

26194 None.

26195 APPLICATION USAGE

26196 The POSIX Realtime Extension defines alternative interfaces for interprocess communication
26197 (IPC). Application developers who need to use IPC should design their applications so that
26198 modules using the IPC routines described in Section 2.7 (on page 39) can be easily modified to
26199 use the alternative interfaces.

26200 RATIONALE

26201 None.

26202 FUTURE DIRECTIONS

26203 None.

26204 SEE ALSO

26205 Section 2.7 (on page 39), Section 2.8 (on page 41), *mq_close()*, *mq_getattr()*, *mq_notify()*,
26206 *mq_open()*, *mq_receive()*, *mq_send()*, *mq_setattr()*, *mq_unlink()*, *msgget()*, *msgrcv()*, *msgsnd()*, the
26207 Base Definitions volume of IEEE Std 1003.1-2001, <sys/msg.h>

26208 CHANGE HISTORY

26209 First released in Issue 2. Derived from Issue 2 of the SVID.

26210 Issue 5

26211 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
26212 DIRECTIONS to a new APPLICATION USAGE section.

26213 NAME

26214 msgget — get the XSI message queue identifier

26215 SYNOPSIS

26216 XSI

```
#include <sys/msg.h>
```

26217

```
int msgget(key_t key, int msgflg);
```

26218

26219 DESCRIPTION

26220 The *msgget()* function operates on XSI message queues (see the Base Definitions volume of
 26221 IEEE Std 1003.1-2001, Section 3.224, Message Queue). It is unspecified whether this function
 26222 interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on
 26223 page 41).

26224 The *msgget()* function shall return the message queue identifier associated with the argument
 26225 *key*.

26226 A message queue identifier, associated message queue, and data structure (see <sys/msg.h>),
 26227 shall be created for the argument *key* if one of the following is true:

- 26228 • The argument *key* is equal to `IPC_PRIVATE`.
- 26229 • The argument *key* does not already have a message queue identifier associated with it, and
 26230 (`msgflg & IPC_CREAT`) is non-zero.

26231 Upon creation, the data structure associated with the new message queue identifier shall be
 26232 initialized as follows:

- 26233 • `msg_perm.cuid`, `msg_perm.uid`, `msg_perm.cgid`, and `msg_perm.gid` shall be set equal to the
 26234 effective user ID and effective group ID, respectively, of the calling process.
- 26235 • The low-order 9 bits of `msg_perm.mode` shall be set equal to the low-order 9 bits of *msgflg*.
- 26236 • `msg_qnum`, `msg_lspid`, `msg_lrpip`, `msg_stime`, and `msg_rtime` shall be set equal to 0.
- 26237 • `msg_ctime` shall be set equal to the current time.
- 26238 • `msg_qbytes` shall be set equal to the system limit.

26239 RETURN VALUE

26240 Upon successful completion, *msgget()* shall return a non-negative integer, namely a message
 26241 queue identifier. Otherwise, it shall return `-1` and set *errno* to indicate the error.

26242 ERRORS

26243 The *msgget()* function shall fail if:

- | | | |
|-------|----------|--|
| 26244 | [EACCES] | A message queue identifier exists for the argument <i>key</i> , but operation |
| 26245 | | permission as specified by the low-order 9 bits of <i>msgflg</i> would not be granted; |
| 26246 | | see Section 2.7 (on page 39). |
| 26247 | [EEXIST] | A message queue identifier exists for the argument <i>key</i> but <code>((msgflg &</code> |
| 26248 | | <code>IPC_CREAT) && (msgflg & IPC_EXCL)</code> is non-zero. |
| 26249 | [ENOENT] | A message queue identifier does not exist for the argument <i>key</i> and <code>(msgflg &</code> |
| 26250 | | <code>IPC_CREAT)</code> is 0. |
| 26251 | [ENOSPC] | A message queue identifier is to be created but the system-imposed limit on |
| 26252 | | the maximum number of allowed message queue identifiers system-wide |
| 26253 | | would be exceeded. |

26254 EXAMPLES

26255 None.

26256 APPLICATION USAGE

26257 The POSIX Realtime Extension defines alternative interfaces for interprocess communication
26258 (IPC). Application developers who need to use IPC should design their applications so that
26259 modules using the IPC routines described in Section 2.7 (on page 39) can be easily modified to
26260 use the alternative interfaces.

26261 RATIONALE

26262 None.

26263 FUTURE DIRECTIONS

26264 None.

26265 SEE ALSO

26266 Section 2.7 (on page 39), Section 2.8 (on page 41), *mq_close()*, *mq_getattr()*, *mq_notify()*,
26267 *mq_open()*, *mq_receive()*, *mq_send()*, *mq_setattr()*, *mq_unlink()*, *msgctl()*, *msgrcv()*, *msgsnd()*, the
26268 Base Definitions volume of IEEE Std 1003.1-2001, <sys/msg.h>

26269 CHANGE HISTORY

26270 First released in Issue 2. Derived from Issue 2 of the SVID.

26271 Issue 5

26272 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
26273 DIRECTIONS to a new APPLICATION USAGE section.

26274 NAME

26275 msgrcv — XSI message receive operation

26276 SYNOPSIS

26277 XSI

```
#include <sys/msg.h>
```

```
26278 ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
26279               int msgflg);
26280
```

26281 DESCRIPTION

26282 The *msgrcv()* function operates on XSI message queues (see the Base Definitions volume of
 26283 IEEE Std 1003.1-2001, Section 3.224, Message Queue). It is unspecified whether this function
 26284 interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on
 26285 page 41).

26286 The *msgrcv()* function shall read a message from the queue associated with the message queue
 26287 identifier specified by *msqid* and place it in the user-defined buffer pointed to by *msgp*.

26288 The application shall ensure that the argument *msgp* points to a user-defined buffer that contains
 26289 first a field of type **long** specifying the type of the message, and then a data portion that holds
 26290 the data bytes of the message. The structure below is an example of what this user-defined
 26291 buffer might look like:

```
26292 struct mymsg {
26293     long    mtype;      /* Message type. */
26294     char    mtext[1];  /* Message text. */
26295 }
```

26296 The structure member *mtype* is the received message's type as specified by the sending process.

26297 The structure member *mtext* is the text of the message.

26298 The argument *msgsz* specifies the size in bytes of *mtext*. The received message shall be truncated
 26299 to *msgsz* bytes if it is larger than *msgsz* and (*msgflg* & MSG_NOERROR) is non-zero. The
 26300 truncated part of the message shall be lost and no indication of the truncation shall be given to
 26301 the calling process.

26302 If the value of *msgsz* is greater than {SSIZE_MAX}, the result is implementation-defined.

26303 The argument *msgtyp* specifies the type of message requested as follows:

- 26304 • If *msgtyp* is 0, the first message on the queue shall be received.
- 26305 • If *msgtyp* is greater than 0, the first message of type *msgtyp* shall be received.
- 26306 • If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the
 26307 absolute value of *msgtyp* shall be received.

26308 The argument *msgflg* specifies the action to be taken if a message of the desired type is not on the
 26309 queue. These are as follows:

- 26310 • If (*msgflg* & IPC_NOWAIT) is non-zero, the calling thread shall return immediately with a
 26311 return value of -1 and *errno* set to [ENOMSG].
- 26312 • If (*msgflg* & IPC_NOWAIT) is 0, the calling thread shall suspend execution until one of the
 26313 following occurs:
 - 26314 — A message of the desired type is placed on the queue.
 - 26315 — The message queue identifier *msqid* is removed from the system; when this occurs, *errno*
 26316 shall be set equal to [EIDRM] and -1 shall be returned.

26317 — The calling thread receives a signal that is to be caught; in this case a message is not
 26318 received and the calling thread resumes execution in the manner prescribed in *sigaction()*.

26319 Upon successful completion, the following actions are taken with respect to the data structure
 26320 associated with *msqid*:

- 26321 • **msg_qnum** shall be decremented by 1.
- 26322 • **msg_lrpid** shall be set equal to the process ID of the calling process.
- 26323 • **msg_rtime** shall be set equal to the current time.

26324 RETURN VALUE

26325 Upon successful completion, *msgrcv()* shall return a value equal to the number of bytes actually
 26326 placed into the buffer *mtext*. Otherwise, no message shall be received, *msgrcv()* shall return
 26327 (**ssize_t**)−1, and *errno* shall be set to indicate the error.

26328 ERRORS

26329 The *msgrcv()* function shall fail if:

- | | | |
|-------|----------|---|
| 26330 | [E2BIG] | The value of <i>mtext</i> is greater than <i>msgsz</i> and (<i>msgflg</i> & MSG_NOERROR) is 0. |
| 26331 | [EACCES] | Operation permission is denied to the calling process; see Section 2.7 (on page 39). |
| 26332 | | |
| 26333 | [EIDRM] | The message queue identifier <i>msqid</i> is removed from the system. |
| 26334 | [EINTR] | The <i>msgrcv()</i> function was interrupted by a signal. |
| 26335 | [EINVAL] | <i>msqid</i> is not a valid message queue identifier. |
| 26336 | [ENOMSG] | The queue does not contain a message of the desired type and (<i>msgflg</i> & IPC_NOWAIT) is non-zero. |
| 26337 | | |

26338 EXAMPLES

26339 Receiving a Message

26340 The following example receives the first message on the queue (based on the value of the *msgtyp*
 26341 argument, 0). The queue is identified by the *msqid* argument (assuming that the value has
 26342 previously been set). This call specifies that an error should be reported if no message is
 26343 available, but not if the message is too large. The message size is calculated directly using the
 26344 *sizeof* operator.

```

26345 #include <sys/msg.h>
26346 ...
26347 int result;
26348 int msqid;
26349 struct message {
26350     long type;
26351     char text[20];
26352 } msg;
26353 long msgtyp = 0;
26354 ...
26355 result = msgrcv(msqid, (void *) &msg, sizeof(msg.text),
26356                 msgtyp, MSG_NOERROR | IPC_NOWAIT);

```


26357 APPLICATION USAGE

26358 The POSIX Realtime Extension defines alternative interfaces for interprocess communication
26359 (IPC). Application developers who need to use IPC should design their applications so that
26360 modules using the IPC routines described in Section 2.7 (on page 39) can be easily modified to
26361 use the alternative interfaces.

26362 RATIONALE

26363 None.

26364 FUTURE DIRECTIONS

26365 None.

26366 SEE ALSO

26367 Section 2.7 (on page 39), Section 2.8 (on page 41), *mq_close()*, *mq_getattr()*, *mq_notify()*,
26368 *mq_open()*, *mq_receive()*, *mq_send()*, *mq_setattr()*, *mq_unlink()*, *msgctl()*, *msgget()*, *msgsnd()*,
26369 *sigaction()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/msg.h>

26370 CHANGE HISTORY

26371 First released in Issue 2. Derived from Issue 2 of the SVID.

26372 Issue 5

26373 The type of the return value is changed from **int** to **ssize_t**, and a warning is added to the
26374 DESCRIPTION about values of *msgsz* larger the {SSIZE_MAX}.

26375 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
26376 DIRECTIONS to the APPLICATION USAGE section.

26377 Issue 6

26378 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

26379 **NAME**

26380 msgsnd — XSI message send operation

26381 **SYNOPSIS**26382 XSI

```
#include <sys/msg.h>
```

26383

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

26384

26385 **DESCRIPTION**

26386 The *msgsnd()* function operates on XSI message queues (see the Base Definitions volume of
 26387 IEEE Std 1003.1-2001, Section 3.224, Message Queue). It is unspecified whether this function
 26388 interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on
 26389 page 41).

26390 The *msgsnd()* function shall send a message to the queue associated with the message queue
 26391 identifier specified by *msqid*.

26392 The application shall ensure that the argument *msgp* points to a user-defined buffer that contains
 26393 first a field of type **long** specifying the type of the message, and then a data portion that holds
 26394 the data bytes of the message. The structure below is an example of what this user-defined
 26395 buffer might look like:

```
26396 struct mymsg {
26397     long    mtype;          /* Message type. */
26398     char    mtext[1];       /* Message text. */
26399 }
```

26400 The structure member *mtype* is a non-zero positive type **long** that can be used by the receiving
 26401 process for message selection.

26402 The structure member *mtext* is any text of length *msgsz* bytes. The argument *msgsz* can range
 26403 from 0 to a system-imposed maximum.

26404 The argument *msgflg* specifies the action to be taken if one or more of the following is true:

- 26405 • The number of bytes already on the queue is equal to **msg_qbytes**; see **<sys/msg.h>**.
- 26406 • The total number of messages on all queues system-wide is equal to the system-imposed
 26407 limit.

26408 These actions are as follows:

- 26409 • If (*msgflg* & **IPC_NOWAIT**) is non-zero, the message shall not be sent and the calling thread
 26410 shall return immediately.
- 26411 • If (*msgflg* & **IPC_NOWAIT**) is 0, the calling thread shall suspend execution until one of the
 26412 following occurs:
 - 26413 — The condition responsible for the suspension no longer exists, in which case the message
 26414 is sent.
 - 26415 — The message queue identifier *msqid* is removed from the system; when this occurs, *errno*
 26416 shall be set equal to **[EIDRM]** and **-1** shall be returned.
 - 26417 — The calling thread receives a signal that is to be caught; in this case the message is not
 26418 sent and the calling thread resumes execution in the manner prescribed in *sigaction()*.

26419 Upon successful completion, the following actions are taken with respect to the data structure
 26420 associated with *msqid*; see **<sys/msg.h>**:

- 26421 • **msg_qnum** shall be incremented by 1.
- 26422 • **msg_lspid** shall be set equal to the process ID of the calling process.
- 26423 • **msg_stime** shall be set equal to the current time.

26424 RETURN VALUE

26425 Upon successful completion, *msgsnd()* shall return 0; otherwise, no message shall be sent,
 26426 *msgsnd()* shall return -1, and *errno* shall be set to indicate the error.

26427 ERRORS

26428 The *msgsnd()* function shall fail if:

- | | | |
|-------------------------|----------|---|
| 26429
26430 | [EACCES] | Operation permission is denied to the calling process; see Section 2.7 (on page 39). |
| 26431
26432 | [EAGAIN] | The message cannot be sent for one of the reasons cited above and (<i>msgflg</i> & <i>IPC_NOWAIT</i>) is non-zero. |
| 26433 | [EIDRM] | The message queue identifier <i>msqid</i> is removed from the system. |
| 26434 | [EINTR] | The <i>msgsnd()</i> function was interrupted by a signal. |
| 26435
26436
26437 | [EINVAL] | The value of <i>msqid</i> is not a valid message queue identifier, or the value of <i>mtype</i> is less than 1; or the value of <i>msgsz</i> is less than 0 or greater than the system-imposed limit. |

26438 EXAMPLES**26439 Sending a Message**

26440 The following example sends a message to the queue identified by the *msqid* argument
 26441 (assuming that value has previously been set). This call specifies that an error should be
 26442 reported if no message is available. The message size is calculated directly using the *sizeof*
 26443 operator.

```

26444     #include <sys/msg.h>
26445     ...
26446     int result;
26447     int msqid;
26448     struct message {
26449         long type;
26450         char text[20];
26451     } msg;

26452     msg.type = 1;
26453     strcpy(msg.text, "This is message 1");
26454     ...
26455     result = msgsnd(msqid, (void *) &msg, sizeof(msg.text), IPC_NOWAIT);
  
```

26456 APPLICATION USAGE

26457 The POSIX Realtime Extension defines alternative interfaces for interprocess communication
 26458 (IPC). Application developers who need to use IPC should design their applications so that
 26459 modules using the IPC routines described in Section 2.7 (on page 39) can be easily modified to
 26460 use the alternative interfaces.

26461 RATIONALE

26462 None.

26463 FUTURE DIRECTIONS

26464 None.

26465 SEE ALSO

26466 Section 2.7 (on page 39), Section 2.8 (on page 41), *mq_close()*, *mq_getattr()*, *mq_notify()*,
26467 *mq_open()*, *mq_receive()*, *mq_send()*, *mq_setattr()*, *mq_unlink()*, *msgctl()*, *msgget()*, *msgrcv()*,
26468 *sigaction()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/msg.h>

26469 CHANGE HISTORY

26470 First released in Issue 2. Derived from Issue 2 of the SVID.

26471 Issue 5

26472 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
26473 DIRECTIONS to a new APPLICATION USAGE section.

26474 Issue 6

26475 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

26476 **NAME**

26477 msync — synchronize memory with physical storage

26478 **SYNOPSIS**

26479 MF SIO #include <sys/mman.h>

26480 int msync(void *addr, size_t len, int flags);

26481

26482 **DESCRIPTION**

26483 The *msync()* function shall write all modified data to permanent storage locations, if any, in
 26484 those whole pages containing any part of the address space of the process starting at address
 26485 *addr* and continuing for *len* bytes. If no such storage exists, *msync()* need not have any effect. If
 26486 requested, the *msync()* function shall then invalidate cached copies of data.

26487 The implementation shall require that *addr* be a multiple of the page size as returned by
 26488 *sysconf()*.

26489 For mappings to files, the *msync()* function shall ensure that all write operations are completed
 26490 as defined for synchronized I/O data integrity completion. It is unspecified whether the
 26491 implementation also writes out other file attributes. When the *msync()* function is called on
 26492 MAP_PRIVATE mappings, any modified data shall not be written to the underlying object and
 26493 shall not cause such data to be made visible to other processes. It is unspecified whether data in
 26494 SHM|TYM MAP_PRIVATE mappings has any permanent storage locations. The effect of *msync()* on a
 26495 shared memory object or a typed memory object is unspecified. The behavior of this function is
 26496 unspecified if the mapping was not established by a call to *mmap()*.

26497 The *flags* argument is constructed from the bitwise-inclusive OR of one or more of the following
 26498 flags defined in the <sys/mman.h> header:

26499

26500

26501

26502

26503

Symbolic Constant	Description
MS_ASYNC	Perform asynchronous writes.
MS_SYNC	Perform synchronous writes.
MS_INVALIDATE	Invalidate cached data.

26504 When MS_ASYNC is specified, *msync()* shall return immediately once all the write operations
 26505 are initiated or queued for servicing; when MS_SYNC is specified, *msync()* shall not return until
 26506 all write operations are completed as defined for synchronized I/O data integrity completion.
 26507 Either MS_ASYNC or MS_SYNC is specified, but not both.

26508 When MS_INVALIDATE is specified, *msync()* shall invalidate all cached copies of mapped data
 26509 that are inconsistent with the permanent storage locations such that subsequent references shall
 26510 obtain data that was consistent with the permanent storage locations sometime between the call
 26511 to *msync()* and the first subsequent memory reference to the data.

26512 If *msync()* causes any write to a file, the file's *st_ctime* and *st_mtime* fields shall be marked for
 26513 update.

26514 **RETURN VALUE**

26515 Upon successful completion, *msync()* shall return 0; otherwise, it shall return -1 and set *errno* to
 26516 indicate the error.

26517 **ERRORS**26518 The *msync()* function shall fail if:

26519 [EBUSY] Some or all of the addresses in the range starting at *addr* and continuing for *len*
 26520 bytes are locked, and MS_INVALIDATE is specified.

- 26521 [EINVAL] The value of *flags* is invalid.
- 26522 [EINVAL] The value of *addr* is not a multiple of the page size {PAGESIZE}.
- 26523 [ENOMEM] The addresses in the range starting at *addr* and continuing for *len* bytes are
- 26524 outside the range allowed for the address space of a process or specify one or
- 26525 more pages that are not mapped.

26526 EXAMPLES

26527 None.

26528 APPLICATION USAGE

26529 The *msync()* function is only supported if the Memory Mapped Files option and the

26530 Synchronized Input and Output option are supported, and thus need not be available on all

26531 implementations.

26532 The *msync()* function should be used by programs that require a memory object to be in a

26533 known state; for example, in building transaction facilities.

26534 Normal system activity can cause pages to be written to disk. Therefore, there are no guarantees

26535 that *msync()* is the only control over when pages are or are not written to disk.

26536 RATIONALE

26537 The *msync()* function writes out data in a mapped region to the permanent storage for the

26538 underlying object. The call to *msync()* ensures data integrity of the file.

26539 After the data is written out, any cached data may be invalidated if the MS_INVALIDATE flag

26540 was specified. This is useful on systems that do not support read/write consistency.

26541 FUTURE DIRECTIONS

26542 None.

26543 SEE ALSO

26544 *mmap()*, *sysconf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/mman.h>

26545 CHANGE HISTORY

26546 First released in Issue 4, Version 2.

26547 Issue 5

26548 Moved from X/OPEN UNIX extension to BASE.

26549 Aligned with *msync()* in the POSIX Realtime Extension as follows:

- 26550 • The DESCRIPTION is extensively reworded.
- 26551 • [EBUSY] and a new form of [EINVAL] are added as mandatory error conditions.

26552 Issue 6

26553 The *msync()* function is marked as part of the Memory Mapped Files and Synchronized Input

26554 and Output options.

26555 The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- 26556 • The [EBUSY] mandatory error condition is added.

26557 The following new requirements on POSIX implementations derive from alignment with the

26558 Single UNIX Specification:

- 26559 • The DESCRIPTION is updated to state that implementations require *addr* to be a multiple of
- 26560 the page size.
- 26561 • The second [EINVAL] error condition is made mandatory.

26562 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding reference to
26563 typed memory objects.

26564 **NAME**

26565 munlock — unlock a range of process address space

26566 **SYNOPSIS**

26567 MLR #include <sys/mman.h>

26568 int munlock(const void *addr, size_t len);

26569

26570 **DESCRIPTION**26571 Refer to *mlock()*.

26572 NAME

26573 munlockall — unlock the address space of a process

26574 SYNOPSIS

26575 ML #include <sys/mman.h>

26576 int munlockall(void);

26577

26578 DESCRIPTION

26579 Refer to *mlockall()*.

26580 **NAME**

26581 munmap — unmap pages of memory

26582 **SYNOPSIS**

26583 MF|SHM #include <sys/mman.h>

26584 int munmap(void *addr, size_t len);

26585

26586 **DESCRIPTION**

26587 The *munmap()* function shall remove any mappings for those entire pages containing any part of
 26588 the address space of the process starting at *addr* and continuing for *len* bytes. Further references
 26589 to these pages shall result in the generation of a SIGSEGV signal to the process. If there are no
 26590 mappings in the specified address range, then *munmap()* has no effect.

26591 The implementation shall require that *addr* be a multiple of the page size {PAGESIZE}.

26592 If a mapping to be removed was private, any modifications made in this address range shall be
 26593 discarded.

26594 ML|MLR Any memory locks (see *mlock()* and *mlockall()*) associated with this address range shall be
 26595 removed, as if by an appropriate call to *munlock()*.

26596 TYM If a mapping removed from a typed memory object causes the corresponding address range of
 26597 the memory pool to be inaccessible by any process in the system except through allocatable
 26598 mappings (that is, mappings of typed memory objects opened with the
 26599 POSIX_TYPED_MEM_MAP_ALLOCATABLE flag), then that range of the memory pool shall
 26600 become deallocated and may become available to satisfy future typed memory allocation
 26601 requests.

26602 A mapping removed from a typed memory object opened with the
 26603 POSIX_TYPED_MEM_MAP_ALLOCATABLE flag shall not affect in any way the availability of
 26604 that typed memory for allocation.

26605 The behavior of this function is unspecified if the mapping was not established by a call to
 26606 *mmap()*.

26607 **RETURN VALUE**

26608 Upon successful completion, *munmap()* shall return 0; otherwise, it shall return -1 and set *errno*
 26609 to indicate the error.

26610 **ERRORS**

26611 The *munmap()* function shall fail if:

26612 [EINVAL] Addresses in the range [*addr*,*addr*+*len*) are outside the valid range for the
 26613 address space of a process.

26614 [EINVAL] The *len* argument is 0.

26615 [EINVAL] The *addr* argument is not a multiple of the page size as returned by *sysconf()*.

26616 **EXAMPLES**

26617 None.

26618 **APPLICATION USAGE**

26619 The *munmap()* function is only supported if the Memory Mapped Files option or the Shared
 26620 Memory Objects option is supported.

26621 **RATIONALE**26622 The *munmap()* function corresponds to SVR4, just as the *mmap()* function does.

26623 It is possible that an application has applied process memory locking to a region that contains
 26624 shared memory. If this has occurred, the *munmap()* call ignores those locks and, if necessary,
 26625 causes those locks to be removed.

26626 **FUTURE DIRECTIONS**

26627 None.

26628 **SEE ALSO**

26629 *mlock()*, *mlockall()*, *mmap()*, *posix_typed_mem_open()*, *sysconf()*, the Base Definitions volume of
 26630 IEEE Std 1003.1-2001, <signal.h>, <sys/mman.h>

26631 **CHANGE HISTORY**

26632 First released in Issue 4, Version 2.

26633 **Issue 5**

26634 Moved from X/OPEN UNIX extension to BASE.

26635 Aligned with *munmap()* in the POSIX Realtime Extension as follows:

- 26636 • The DESCRIPTION is extensively reworded.
- 26637 • The SIGBUS error is no longer permitted to be generated.

26638 **Issue 6**

26639 The *munmap()* function is marked as part of the Memory Mapped Files and Shared Memory
 26640 Objects option.

26641 The following new requirements on POSIX implementations derive from alignment with the
 26642 Single UNIX Specification:

- 26643 • The DESCRIPTION is updated to state that implementations require *addr* to be a multiple of
 26644 the page size.
- 26645 • The [EINVAL] error conditions are added.

26646 The following changes are made for alignment with IEEE Std 1003.1j-2000:

- 26647 • Semantics for typed memory objects are added to the DESCRIPTION.
- 26648 • The *posix_typed_mem_open()* function is added to the SEE ALSO section.

26649 **NAME**

26650 nan, nanf, nanl — return quiet NaN

26651 **SYNOPSIS**

26652 #include <math.h>

26653 double nan(const char *tagp);

26654 float nanf(const char *tagp);

26655 long double nanl(const char *tagp);

26656 **DESCRIPTION**

26657 cx The functionality described on this reference page is aligned with the ISO C standard. Any
 26658 conflict between the requirements described here and the ISO C standard is unintentional. This
 26659 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

26660 The function call *nan*("n-char-sequence") shall be equivalent to:

26661 strtod("NAN(n-char-sequence)", (char **) NULL);

26662 The function call *nan*("") shall be equivalent to:

26663 strtod("NAN()", (char **) NULL)

26664 If *tagp* does not point to an *n-char* sequence or an empty string, the function call shall be
 26665 equivalent to:

26666 strtod("NAN", (char **) NULL)

26667 Function calls to *nanf*() and *nanl*() are equivalent to the corresponding function calls to *strtof*()
 26668 and *strtold*().

26669 **RETURN VALUE**26670 These functions shall return a quiet NaN, if available, with content indicated through *tagp*.

26671 If the implementation does not support quiet NaNs, these functions shall return zero.

26672 **ERRORS**

26673 No errors are defined.

26674 **EXAMPLES**

26675 None.

26676 **APPLICATION USAGE**

26677 None.

26678 **RATIONALE**

26679 None.

26680 **FUTURE DIRECTIONS**

26681 None.

26682 **SEE ALSO**26683 *strtod*(), *strtold*(), the Base Definitions volume of IEEE Std 1003.1-2001, <math.h>26684 **CHANGE HISTORY**

26685 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

26686 **NAME**26687 nanosleep — high resolution sleep (**REALTIME**)26688 **SYNOPSIS**

26689 TMR #include <time.h>

26690 int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);

26691

26692 **DESCRIPTION**

26693 The *nanosleep()* function shall cause the current thread to be suspended from execution until
 26694 either the time interval specified by the *rqtp* argument has elapsed or a signal is delivered to the
 26695 calling thread, and its action is to invoke a signal-catching function or to terminate the process.
 26696 The suspension time may be longer than requested because the argument value is rounded up to
 26697 an integer multiple of the sleep resolution or because of the scheduling of other activity by the
 26698 system. But, except for the case of being interrupted by a signal, the suspension time shall not be
 26699 less than the time specified by *rqtp*, as measured by the system clock **CLOCK_REALTIME**.

26700 The use of the *nanosleep()* function has no effect on the action or blockage of any signal.26701 **RETURN VALUE**26702 If the *nanosleep()* function returns because the requested time has elapsed, its return value shall
26703 be zero.

26704 If the *nanosleep()* function returns because it has been interrupted by a signal, it shall return a
 26705 value of **-1** and set *errno* to indicate the interruption. If the *rmtp* argument is non-NULL, the
 26706 **timespec** structure referenced by it is updated to contain the amount of time remaining in the
 26707 interval (the requested time minus the time actually slept). If the *rmtp* argument is NULL, the
 26708 remaining time is not returned.

26709 If *nanosleep()* fails, it shall return a value of **-1** and set *errno* to indicate the error.26710 **ERRORS**26711 The *nanosleep()* function shall fail if:26712 [EINTR] The *nanosleep()* function was interrupted by a signal.

26713 [EINVAL] The *rqtp* argument specified a nanosecond value less than zero or greater than
 26714 or equal to 1 000 million.

26715 **EXAMPLES**

26716 None.

26717 **APPLICATION USAGE**

26718 None.

26719 **RATIONALE**

26720 It is common to suspend execution of a process for an interval in order to poll the status of a
 26721 non-interrupting function. A large number of actual needs can be met with a simple extension to
 26722 *sleep()* that provides finer resolution.

26723 In the POSIX.1-1990 standard and SVR4, it is possible to implement such a routine, but the
 26724 frequency of wakeup is limited by the resolution of the *alarm()* and *sleep()* functions. In 4.3 BSD,
 26725 it is possible to write such a routine using no static storage and reserving no system facilities.
 26726 Although it is possible to write a function with similar functionality to *sleep()* using the
 26727 remainder of the *timer_**() functions, such a function requires the use of signals and the
 26728 reservation of some signal number. This volume of IEEE Std 1003.1-2001 requires that
 26729 *nanosleep()* be non-intrusive of the signals function.

26730 The *nanosleep()* function shall return a value of 0 on success and –1 on failure or if interrupted.
26731 This latter case is different from *sleep()*. This was done because the remaining time is returned
26732 via an argument structure pointer, *rmtp*, instead of as the return value.

26733 **FUTURE DIRECTIONS**

26734 None.

26735 **SEE ALSO**

26736 *sleep()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**time.h**>

26737 **CHANGE HISTORY**

26738 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

26739 **Issue 6**

26740 The *nanosleep()* function is marked as part of the Timers option.

26741 The [ENOSYS] error condition has been removed as stubs need not be provided if an
26742 implementation does not support the Timers option.

26743 **NAME**

26744 nearbyint, nearbyintf, nearbyintl — floating-point rounding functions

26745 **SYNOPSIS**

26746 #include <math.h>

26747 double nearbyint(double x);

26748 float nearbyintf(float x);

26749 long double nearbyintl(long double x);

26750 **DESCRIPTION**

26751 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 26752 conflict between the requirements described here and the ISO C standard is unintentional. This
 26753 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

26754 These functions shall round their argument to an integer value in floating-point format, using
 26755 the current rounding direction and without raising the inexact floating-point exception.

26756 An application wishing to check for error situations should set *errno* to zero and call
 26757 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 26758 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 26759 zero, an error has occurred.

26760 **RETURN VALUE**

26761 Upon successful completion, these functions shall return the rounded integer value.

26762 MX If *x* is NaN, a NaN shall be returned.26763 If *x* is ± 0 , ± 0 shall be returned.26764 If *x* is $\pm \text{Inf}$, *x* shall be returned.

26765 XSI If the correct value would cause overflow, a range error shall occur and *nearbyint*(), *nearbyintf*(),
 26766 and *nearbyintl*() shall return the value of the macro $\pm \text{HUGE_VAL}$, $\pm \text{HUGE_VALF}$, and
 26767 $\pm \text{HUGE_VALL}$ (with the same sign as *x*), respectively.

26768 **ERRORS**

26769 These functions shall fail if:

26770 XSI **Range Error** The result would cause an overflow.

26771 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 26772 then *errno* shall be set to [ERANGE]. If the integer expression
 26773 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow
 26774 floating-point exception shall be raised.

26775 **EXAMPLES**

26776 None.

26777 **APPLICATION USAGE**

26778 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
 26779 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

26780 **RATIONALE**

26781 None.

26782 **FUTURE DIRECTIONS**

26783 None.

26784 **SEE ALSO**

26785 *feclearexcept()*, *fetestexcept()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18,
26786 Treatment of Error Conditions for Mathematical Functions, <**math.h**>

26787 **CHANGE HISTORY**

26788 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

26789 NAME

26790 nextafter, nextafterf, nextafterl, nexttoward, nexttowardf, nexttowardl — next representable
 26791 floating-point number

26792 SYNOPSIS

26793 #include <math.h>

26794 double nextafter(double x, double y);
 26795 float nextafterf(float x, float y);
 26796 long double nextafterl(long double x, long double y);
 26797 double nexttoward(double x, long double y);
 26798 float nexttowardf(float x, long double y);
 26799 long double nexttowardl(long double x, long double y);

26800 DESCRIPTION

26801 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 26802 conflict between the requirements described here and the ISO C standard is unintentional. This
 26803 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

26804 The *nextafter()*, *nextafterf()*, and *nextafterl()* functions shall compute the next representable
 26805 floating-point value following *x* in the direction of *y*. Thus, if *y* is less than *x*, *nextafter()* shall
 26806 return the largest representable floating-point number less than *x*. The *nextafter()*, *nextafterf()*,
 26807 and *nextafterl()* functions shall return *y* if *x* equals *y*.

26808 The *nexttoward()*, *nexttowardf()*, and *nexttowardl()* functions shall be equivalent to the
 26809 corresponding *nextafter()* functions, except that the second parameter shall have type **long**
 26810 **double** and the functions shall return *y* converted to the type of the function if *x* equals *y*.

26811 An application wishing to check for error situations should set *errno* to zero and call
 26812 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 26813 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 26814 zero, an error has occurred.

26815 RETURN VALUE

26816 Upon successful completion, these functions shall return the next representable floating-point
 26817 value following *x* in the direction of *y*.

26818 If *x*==*y*, *y* (of the type *x*) shall be returned.

26819 If *x* is finite and the correct function value would overflow, a range error shall occur and
 26820 ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL (with the same sign as *x*) shall be returned as
 26821 appropriate for the return type of the function.

26822 MX If *x* or *y* is NaN, a NaN shall be returned.

26823 If *x*!=*y* and the correct function value is subnormal, zero, or underflows, a range error shall
 26824 occur, and either the correct function value (if representable) or 0.0 shall be returned.

26825 ERRORS

26826 These functions shall fail if:

26827 Range Error The correct value overflows.

26828 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 26829 then *errno* shall be set to [ERANGE]. If the integer expression
 26830 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow
 26831 floating-point exception shall be raised.

26832 MX Range Error The correct value is subnormal or underflows.

26833 If the integer expression (`math_errhandling & MATH_ERRNO`) is non-zero,
26834 then *errno* shall be set to [ERANGE]. If the integer expression
26835 (`math_errhandling & MATH_ERREXCEPT`) is non-zero, then the underflow
26836 floating-point exception shall be raised.

26837 EXAMPLES

26838 None.

26839 APPLICATION USAGE

26840 On error, the expressions (`math_errhandling & MATH_ERRNO`) and (`math_errhandling &`
26841 `MATH_ERREXCEPT`) are independent of each other, but at least one of them must be non-zero.

26842 RATIONALE

26843 None.

26844 FUTURE DIRECTIONS

26845 None.

26846 SEE ALSO

26847 *feclearexcept()*, *fetetestexcept()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18,
26848 Treatment of Error Conditions for Mathematical Functions, <**math.h**>

26849 CHANGE HISTORY

26850 First released in Issue 4, Version 2.

26851 Issue 5

26852 Moved from X/OPEN UNIX extension to BASE.

26853 Issue 6

26854 The *nextafter()* function is no longer marked as an extension.

26855 The *nextafterf()*, *nextafterl()*, *nexttoward()*, *nexttowardf()*, and *nexttowardl()* functions are added
26856 for alignment with the ISO/IEC 9899:1999 standard.

26857 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
26858 revised to align with the ISO/IEC 9899:1999 standard.

26859 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
26860 marked.

26861 NAME

26862 nftw — walk a file tree

26863 SYNOPSIS

26864 XSI `#include <ftw.h>`

```
26865 int nftw(const char *path, int (*fn)(const char *,
26866     const struct stat *, int, struct FTW *), int depth, int flags);
26867
```

26868 DESCRIPTION

26869 The *nftw()* function shall recursively descend the directory hierarchy rooted in *path*. The *nftw()*
 26870 function has a similar effect to *ftw()* except that it takes an additional argument *flags*, which is a
 26871 bitwise-inclusive OR of zero or more of the following flags:

26872 **FTW_CHDIR** If set, *nftw()* shall change the current working directory to each directory as it
 26873 reports files in that directory. If clear, *nftw()* shall not change the current
 26874 working directory.

26875 **FTW_DEPTH** If set, *nftw()* shall report all files in a directory before reporting the directory
 26876 itself. If clear, *nftw()* shall report any directory before reporting the files in that
 26877 directory.

26878 **FTW_MOUNT** If set, *nftw()* shall only report files in the same file system as *path*. If clear,
 26879 *nftw()* shall report all files encountered during the walk.

26880 **FTW_PHYS** If set, *nftw()* shall perform a physical walk and shall not follow symbolic links.

26881 If **FTW_PHYS** is clear and **FTW_DEPTH** is set, *nftw()* shall follow links instead of reporting
 26882 them, but shall not report any directory that would be a descendant of itself. If **FTW_PHYS** is
 26883 clear and **FTW_DEPTH** is clear, *nftw()* shall follow links instead of reporting them, but shall not
 26884 report the contents of any directory that would be a descendant of itself.

26885 At each file it encounters, *nftw()* shall call the user-supplied function *fn* with four arguments:

- 26886 • The first argument is the pathname of the object.
- 26887 • The second argument is a pointer to the **stat** buffer containing information on the object.
- 26888 • The third argument is an integer giving additional information. Its value is one of the
 26889 following:

26890 **FTW_F** The object is a file.

26891 **FTW_D** The object is a directory.

26892 **FTW_DP** The object is a directory and subdirectories have been visited. (This condition
 26893 shall only occur if the **FTW_DEPTH** flag is included in *flags*.)

26894 **FTW_SL** The object is a symbolic link. (This condition shall only occur if the **FTW_PHYS**
 26895 flag is included in *flags*.)

26896 **FTW_SLN** The object is a symbolic link that does not name an existing file. (This
 26897 condition shall only occur if the **FTW_PHYS** flag is not included in *flags*.)

26898 **FTW_DNR** The object is a directory that cannot be read. The *fn* function shall not be called
 26899 for any of its descendants.

26900 **FTW_NS** The *stat()* function failed on the object because of lack of appropriate
 26901 permission. The **stat** buffer passed to *fn* is undefined. Failure of *stat()* for any
 26902 other reason is considered an error and *nftw()* shall return `-1`.

26903 • The fourth argument is a pointer to an **FTW** structure. The value of **base** is the offset of the
 26904 object's filename in the pathname passed as the first argument to *fn*. The value of **level**
 26905 indicates depth relative to the root of the walk, where the root level is 0.

26906 The results are unspecified if the application-supplied *fn* function does not preserve the current
 26907 working directory.

26908 The argument *depth* sets the maximum number of file descriptors that shall be used by *nftw()*
 26909 while traversing the file tree. At most one file descriptor shall be used for each directory level.

26910 The *nftw()* function need not be reentrant. A function that is not required to be reentrant is not
 26911 required to be thread-safe.

26912 RETURN VALUE

26913 The *nftw()* function shall continue until the first of the following conditions occurs:

- 26914 • An invocation of *fn* shall return a non-zero value, in which case *nftw()* shall return that value.
- 26915 • The *nftw()* function detects an error other than [EACCES] (see FTW_DNR and FTW_NS
 26916 above), in which case *nftw()* shall return -1 and set *errno* to indicate the error.
- 26917 • The tree is exhausted, in which case *nftw()* shall return 0.

26918 ERRORS

26919 The *nftw()* function shall fail if:

26920 [EACCES] Search permission is denied for any component of *path* or read permission is
 26921 denied for *path*, or *fn* returns -1 and does not reset *errno*.

26922 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 26923 argument.

26924 [ENAMETOOLONG]

26925 The length of the *path* argument exceeds {PATH_MAX} or a pathname
 26926 component is longer than {NAME_MAX}.

26927 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

26928 [ENOTDIR] A component of *path* is not a directory.

26929 [EOVERFLOW] A field in the **stat** structure cannot be represented correctly in the current
 26930 programming environment for one or more files found in the file hierarchy.

26931 The *nftw()* function may fail if:

26932 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 26933 resolution of the *path* argument.

26934 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

26935 [ENAMETOOLONG]

26936 Pathname resolution of a symbolic link produced an intermediate result
 26937 whose length exceeds {PATH_MAX}.

26938 [ENFILE] Too many files are currently open in the system.

26939 In addition, *errno* may be set if the function pointed to by *fn* causes *errno* to be set.

26940 EXAMPLES

26941 The following example walks the **/tmp** directory and its subdirectories, calling the *nftw()*
 26942 function for every directory entry, to a maximum of 5 levels deep.

```
26943 #include <ftw.h>
26944 ...
26945 int nftwfunc(const char *, const struct stat *, int, struct FTW *);
26946
26947 int nftwfunc(const char *filename, const struct stat *statptr,
26948             int fileflags, struct FTW *pftw)
26949 {
26950     return 0;
26951 }
26952 ...
26953 char *startpath = "/tmp";
26954 int depth = 5;
26955 int flags = FTW_CHDIR | FTW_DEPTH | FTW_MOUNT;
26956 int ret;
26957
26958 ret = nftw(startpath, nftwfunc, depth, flags);
```

26957 APPLICATION USAGE

26958 None.

26959 RATIONALE

26960 None.

26961 FUTURE DIRECTIONS

26962 None.

26963 SEE ALSO

26964 *lstat()*, *opendir()*, *readdir()*, *stat()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<ftw.h>**

26965 CHANGE HISTORY

26966 First released in Issue 4, Version 2.

26967 Issue 5

26968 Moved from X/OPEN UNIX extension to BASE.

26969 In the DESCRIPTION, the definition of the *depth* argument is clarified.

26970 Issue 6

26971 The Open Group Base Resolution bwg97-003 is applied.

26972 The ERRORS section is updated as follows:

- 26973 • The wording of the mandatory [ELOOP] error condition is updated.
- 26974 • A second optional [ELOOP] error condition is added.
- 26975 • The [EOVERFLOW] mandatory error condition is added.

26976 Text is added to the DESCRIPTION to say that the *nftw()* function need not be reentrant and
 26977 that the results are unspecified if the application-supplied *fn* function does not preserve the
 26978 current working directory.

26979 **NAME**26980 *nice* — change the nice value of a process26981 **SYNOPSIS**26982 XSI `#include <unistd.h>`26983 `int nice(int incr);`

26984

26985 **DESCRIPTION**

26986 The *nice()* function shall add the value of *incr* to the nice value of the calling process. A process' nice value is a non-negative number for which a more positive value shall result in less favorable scheduling.

26989 A maximum nice value of 2*{NZERO}-1 and a minimum nice value of 0 shall be imposed by the system. Requests for values above or below these limits shall result in the nice value being set to the corresponding limit. Only a process with appropriate privileges can lower the nice value.

26992 PS|TPS Calling the *nice()* function has no effect on the priority of processes or threads with policy SCHED_FIFO or SCHED_RR. The effect on processes or threads with other scheduling policies is implementation-defined.

26995 The nice value set with *nice()* shall be applied to the process. If the process is multi-threaded, the nice value shall affect all system scope threads in the process.

26997 As -1 is a permissible return value in a successful situation, an application wishing to check for error situations should set *errno* to 0, then call *nice()*, and if it returns -1, check to see whether *errno* is non-zero.

27000 **RETURN VALUE**

27001 Upon successful completion, *nice()* shall return the new nice value -{NZERO}. Otherwise, -1 shall be returned, the process' nice value shall not be changed, and *errno* shall be set to indicate the error.

27004 **ERRORS**27005 The *nice()* function shall fail if:

27006 [EPERM] The *incr* argument is negative and the calling process does not have appropriate privileges.

27008 **EXAMPLES**27009 **Changing the Nice Value**

27010 The following example adds the value of the *incr* argument, -20, to the nice value of the calling process.

27012 `#include <unistd.h>`27013 `...`27014 `int incr = -20;`27015 `int ret;`27016 `ret = nice(incr);`27017 **APPLICATION USAGE**

27018 None.

27019 RATIONALE

27020 None.

27021 FUTURE DIRECTIONS

27022 None.

27023 SEE ALSO

27024 *getpriority()*, *setpriority()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<limits.h>**,
27025 **<unistd.h>**

27026 CHANGE HISTORY

27027 First released in Issue 1. Derived from Issue 1 of the SVID.

27028 Issue 5

27029 A statement is added to the description indicating the effects of this function on the different
27030 scheduling policies and multi-threaded processes.

27031 **NAME**

27032 nl_langinfo — language information

27033 **SYNOPSIS**27034 XSI `#include <langinfo.h>`27035 `char *nl_langinfo(nl_item item);`

27036

27037 **DESCRIPTION**

27038 The *nl_langinfo()* function shall return a pointer to a string containing information relevant to
 27039 the particular language or cultural area defined in the program's locale (see <langinfo.h>). The
 27040 manifest constant names and values of *item* are defined in <langinfo.h>. For example:

27041 `nl_langinfo(ABDAY_1)`

27042 would return a pointer to the string "Dom" if the identified language was Portuguese, and
 27043 "Sun" if the identified language was English.

27044 Calls to *setlocale()* with a category corresponding to the category of *item* (see <langinfo.h>), or to
 27045 the category *LC_ALL*, may overwrite the array pointed to by the return value.

27046 The *nl_langinfo()* function need not be reentrant. A function that is not required to be reentrant is
 27047 not required to be thread-safe.

27048 **RETURN VALUE**

27049 In a locale where *langinfo* data is not defined, *nl_langinfo()* shall return a pointer to the
 27050 corresponding string in the POSIX locale. In all locales, *nl_langinfo()* shall return a pointer to an
 27051 empty string if *item* contains an invalid setting.

27052 This pointer may point to static data that may be overwritten on the next call.

27053 **ERRORS**

27054 No errors are defined.

27055 **EXAMPLES**27056 **Getting Date and Time Formatting Information**

27057 The following example returns a pointer to a string containing date and time formatting
 27058 information, as defined in the *LC_TIME* category of the current locale.

27059 `#include <time.h>`27060 `#include <langinfo.h>`27061 `...`27062 `strftime(datestring, sizeof(datestring), nl_langinfo(D_T_FMT), tm);`27063 `...`27064 **APPLICATION USAGE**

27065 The array pointed to by the return value should not be modified by the program, but may be
 27066 modified by further calls to *nl_langinfo()*.

27067 **RATIONALE**

27068 None.

27069 **FUTURE DIRECTIONS**

27070 None.

27071 SEE ALSO

27072 *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <**langinfo.h**>,
27073 <**nl_types.h**>

27074 CHANGE HISTORY

27075 First released in Issue 2.

27076 Issue 5

27077 The last paragraph of the DESCRIPTION is moved from the APPLICATION USAGE section.

27078 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

27079 **NAME**

27080 nrnd48 — generate uniformly distributed pseudo-random non-negative long integers

27081 **SYNOPSIS**

27082 xSI #include <stdlib.h>

27083 long nrnd48(unsigned short xsubi[3]);

27084

27085 **DESCRIPTION**

27086 Refer to *drnd48()*.

27087 **NAME**

27088 ntohl, ntohs — convert values between host and network byte order

27089 **SYNOPSIS**

27090 #include <arpa/inet.h>

27091 uint32_t ntohl(uint32_t *netlong*);27092 uint16_t ntohs(uint16_t *netshort*);27093 **DESCRIPTION**27094 Refer to *htonl()*.

27095 **NAME**

27096 open — open a file

27097 **SYNOPSIS**

27098 OH #include <sys/stat.h>

27099 #include <fcntl.h>

27100 int open(const char *path, int oflag, ...);

27101 **DESCRIPTION**

27102 The *open()* function shall establish the connection between a file and a file descriptor. It shall
 27103 create an open file description that refers to a file and a file descriptor that refers to that open file
 27104 description. The file descriptor is used by other I/O functions to refer to that file. The *path*
 27105 argument points to a pathname naming the file.

27106 The *open()* function shall return a file descriptor for the named file that is the lowest file
 27107 descriptor not currently open for that process. The open file description is new, and therefore the
 27108 file descriptor shall not share it with any other process in the system. The FD_CLOEXEC file
 27109 descriptor flag associated with the new file descriptor shall be cleared.

27110 The file offset used to mark the current position within the file shall be set to the beginning of the
 27111 file.

27112 The file status flags and file access modes of the open file description shall be set according to
 27113 the value of *oflag*.

27114 Values for *oflag* are constructed by a bitwise-inclusive OR of flags from the following list,
 27115 defined in <fcntl.h>. Applications shall specify exactly one of the first three values (file access
 27116 modes) below in the value of *oflag*:

27117 O_RDONLY Open for reading only.

27118 O_WRONLY Open for writing only.

27119 O_RDWR Open for reading and writing. The result is undefined if this flag is applied to
 27120 a FIFO.

27121 Any combination of the following may be used:

27122 O_APPEND If set, the file offset shall be set to the end of the file prior to each write.

27123 O_CREAT If the file exists, this flag has no effect except as noted under O_EXCL below.
 27124 Otherwise, the file shall be created; the user ID of the file shall be set to the
 27125 effective user ID of the process; the group ID of the file shall be set to the
 27126 group ID of the file's parent directory or to the effective group ID of the
 27127 process; and the access permission bits (see <sys/stat.h>) of the file mode shall
 27128 be set to the value of the third argument taken as type **mode_t** modified as
 27129 follows: a bitwise AND is performed on the file-mode bits and the
 27130 corresponding bits in the complement of the process' file mode creation mask.
 27131 Thus, all bits in the file mode whose corresponding bit in the file mode
 27132 creation mask is set are cleared. When bits other than the file permission bits
 27133 are set, the effect is unspecified. The third argument does not affect whether
 27134 the file is open for reading, writing, or for both. Implementations shall provide
 27135 a way to initialize the file's group ID to the group ID of the parent directory.
 27136 Implementations may, but need not, provide an implementation-defined way
 27137 to initialize the file's group ID to the effective group ID of the calling process.

27138 SIO O_DSYNC Write I/O operations on the file descriptor shall complete as defined by
 27139 synchronized I/O data integrity completion.

27140	O_EXCL	If O_CREAT and O_EXCL are set, <i>open()</i> shall fail if the file exists. The check for the existence of the file and the creation of the file if it does not exist shall be atomic with respect to other threads executing <i>open()</i> naming the same filename in the same directory with O_EXCL and O_CREAT set. If O_EXCL and O_CREAT are set, and <i>path</i> names a symbolic link, <i>open()</i> shall fail and set <i>errno</i> to [EEXIST], regardless of the contents of the symbolic link. If O_EXCL is set and O_CREAT is not set, the result is undefined.
27141		
27142		
27143		
27144		
27145		
27146		
27147	O_NOCTTY	If set and <i>path</i> identifies a terminal device, <i>open()</i> shall not cause the terminal device to become the controlling terminal for the process.
27148		
27149	O_NONBLOCK	When opening a FIFO with O_RDONLY or O_WRONLY set:
27150		• If O_NONBLOCK is set, an <i>open()</i> for reading-only shall return without delay. An <i>open()</i> for writing-only shall return an error if no process currently has the file open for reading.
27151		
27152		
27153		• If O_NONBLOCK is clear, an <i>open()</i> for reading-only shall block the calling thread until a thread opens the file for writing. An <i>open()</i> for writing-only shall block the calling thread until a thread opens the file for reading.
27154		
27155		
27156		
27157		When opening a block special or character special file that supports non-blocking opens:
27158		
27159		• If O_NONBLOCK is set, the <i>open()</i> function shall return without blocking for the device to be ready or available. Subsequent behavior of the device is device-specific.
27160		
27161		
27162		• If O_NONBLOCK is clear, the <i>open()</i> function shall block the calling thread until the device is ready or available before returning.
27163		
27164		Otherwise, the behavior of O_NONBLOCK is unspecified.
27165 SIO	O_RSYNC	Read I/O operations on the file descriptor shall complete at the same level of integrity as specified by the O_DSYNC and O_SYNC flags. If both O_DSYNC and O_RSYNC are set in <i>oflag</i> , all I/O operations on the file descriptor shall complete as defined by synchronized I/O data integrity completion. If both O_SYNC and O_RSYNC are set in flags, all I/O operations on the file descriptor shall complete as defined by synchronized I/O file integrity completion.
27166		
27167		
27168		
27169		
27170		
27171		
27172 SIO	O_SYNC	Write I/O operations on the file descriptor shall complete as defined by synchronized I/O file integrity completion.
27173		
27174	O_TRUNC	If the file exists and is a regular file, and the file is successfully opened O_RDWR or O_WRONLY, its length shall be truncated to 0, and the mode and owner shall be unchanged. It shall have no effect on FIFO special files or terminal device files. Its effect on other file types is implementation-defined. The result of using O_TRUNC with O_RDONLY is undefined.
27175		
27176		
27177		
27178		
27179		If O_CREAT is set and the file did not previously exist, upon successful completion, <i>open()</i> shall mark for update the <i>st_atime</i> , <i>st_ctime</i> , and <i>st_mtime</i> fields of the file and the <i>st_ctime</i> and <i>st_mtime</i> fields of the parent directory.
27180		
27181		
27182		If O_TRUNC is set and the file did previously exist, upon successful completion, <i>open()</i> shall mark for update the <i>st_ctime</i> and <i>st_mtime</i> fields of the file.
27183		

27184 SIO 27185	If both the O_SYNC and O_DSYNC flags are set, the effect is as if only the O_SYNC flag was set.	
27186 XSR 27187 27188 27189 27190 27191	If <i>path</i> refers to a STREAMS file, <i>oflag</i> may be constructed from O_NONBLOCK OR'ed with either O_RDONLY, O_WRONLY, or O_RDWR. Other flag values are not applicable to STREAMS devices and shall have no effect on them. The value O_NONBLOCK affects the operation of STREAMS drivers and certain functions applied to file descriptors associated with STREAMS files. For STREAMS drivers, the implementation of O_NONBLOCK is device-specific.	
27192 XSI 27193 27194	If <i>path</i> names the master side of a pseudo-terminal device, then it is unspecified whether <i>open()</i> locks the slave side so that it cannot be opened. Conforming applications shall call <i>unlockpt()</i> before opening the slave side.	
27195 27196	The largest value that can be represented correctly in an object of type off_t shall be established as the offset maximum in the open file description.	
27197	RETURN VALUE	
27198 27199 27200	Upon successful completion, the function shall open the file and return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, -1 shall be returned and <i>errno</i> set to indicate the error. No files shall be created or modified if the function returns -1 .	
27201	ERRORS	
27202	The <i>open()</i> function shall fail if:	
27203 27204 27205 27206	[EACCES]	Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by <i>oflag</i> are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created, or O_TRUNC is specified and write permission is denied.
27207	[EEXIST]	O_CREAT and O_EXCL are set, and the named file exists.
27208	[EINTR]	A signal was caught during <i>open()</i> .
27209 SIO	[EINVAL]	The implementation does not support synchronized I/O for this file.
27210 XSR 27211	[EIO]	The <i>path</i> argument names a STREAMS file and a hangup or error occurred during the <i>open()</i> .
27212	[EISDIR]	The named file is a directory and <i>oflag</i> includes O_WRONLY or O_RDWR.
27213 27214	[ELOOP]	A loop exists in symbolic links encountered during resolution of the <i>path</i> argument.
27215	[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
27216 27217 27218	[ENAMETOOLONG]	The length of the <i>path</i> argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
27219	[ENFILE]	The maximum allowable number of files is currently open in the system.
27220 27221 27222	[ENOENT]	O_CREAT is not set and the named file does not exist; or O_CREAT is set and either the path prefix does not exist or the <i>path</i> argument points to an empty string.
27223 XSR 27224	[ENOSR]	The <i>path</i> argument names a STREAMS-based file and the system is unable to allocate a STREAM.
27225 27226	[ENOSPC]	The directory or file system that would contain the new file cannot be expanded, the file does not exist, and O_CREAT is specified.

27227	[ENOTDIR]	A component of the path prefix is not a directory.
27228	[ENXIO]	O_NONBLOCK is set, the named file is a FIFO, O_WRONLY is set, and no
27229		process has the file open for reading.
27230	[ENXIO]	The named file is a character special or block special file, and the device
27231		associated with this special file does not exist.
27232	[EOVERFLOW]	The named file is a regular file and the size of the file cannot be represented
27233		correctly in an object of type <code>off_t</code> .
27234	[EROFS]	The named file resides on a read-only file system and either O_WRONLY,
27235		O_RDWR, O_CREAT (if the file does not exist), or O_TRUNC is set in the <i>oflag</i>
27236		argument.
27237	The <i>open()</i> function may fail if:	
27238 XSI	[EAGAIN]	The <i>path</i> argument names the slave side of a pseudo-terminal device that is
27239		locked.
27240	[EINVAL]	The value of the <i>oflag</i> argument is not valid.
27241	[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during
27242		resolution of the <i>path</i> argument.
27243	[ENAMETOOLONG]	
27244		As a result of encountering a symbolic link in resolution of the <i>path</i> argument,
27245		the length of the substituted pathname string exceeded {PATH_MAX}.
27246 XSR	[ENOMEM]	The <i>path</i> argument names a STREAMS file and the system is unable to allocate
27247		resources.
27248	[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed and <i>oflag</i> is
27249		O_WRONLY or O_RDWR.

27250 EXAMPLES

27251 Opening a File for Writing by the Owner

27252 The following example opens the file `/tmp/file`, either by creating it (if it does not already exist),
 27253 or by truncating its length to 0 (if it does exist). In the former case, if the call creates a new file,
 27254 the access permission bits in the file mode of the file are set to permit reading and writing by the
 27255 owner, and to permit reading only by group members and others.

27256 If the call to *open()* is successful, the file is opened for writing.

```

27257 #include <fcntl.h>
27258 ...
27259 int fd;
27260 mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
27261 char *filename = "/tmp/file";
27262 ...
27263 fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, mode);
27264 ...

```


Opening a File Using an Existence Check

The following example uses the *open()* function to try to create the **LOCKFILE** file and open it for writing. Since the *open()* function specifies the **O_EXCL** flag, the call fails if the file already exists. In that case, the program assumes that someone else is updating the password file and exits.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

#define LOCKFILE "/etc/ptmp"
...
int pfd; /* Integer for file descriptor returned by open() call. */
...
if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
    exit(1);
}
...
```

Opening a File for Writing

The following example opens a file for writing, creating the file if it does not already exist. If the file does exist, the system truncates the file to zero bytes.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

#define LOCKFILE "/etc/ptmp"
...
int pfd;
char filename[PATH_MAX+1];
...
if ((pfd = open(filename, O_WRONLY | O_CREAT | O_TRUNC,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    perror("Cannot open output file\n"); exit(1);
}
...
```

APPLICATION USAGE

None.

RATIONALE

Except as specified in this volume of IEEE Std 1003.1-2001, the flags allowed in *oflag* are not mutually-exclusive and any number of them may be used simultaneously.

Some implementations permit opening FIFOs with **O_RDWR**. Since FIFOs could be implemented in other ways, and since two file descriptors can be used to the same effect, this possibility is left as undefined.

See *getgroups()* about the group of a newly created file.

27310 The use of *open()* to create a regular file is preferable to the use of *creat()*, because the latter is
 27311 redundant and included only for historical reasons.

27312 The use of the O_TRUNC flag on FIFOs and directories (pipes cannot be *open()*-ed) must be
 27313 permissible without unexpected side effects (for example, *creat()* on a FIFO must not remove
 27314 data). Since terminal special files might have type-ahead data stored in the buffer, O_TRUNC
 27315 should not affect their content, particularly if a program that normally opens a regular file
 27316 should open the current controlling terminal instead. Other file types, particularly
 27317 implementation-defined ones, are left implementation-defined.

27318 IEEE Std 1003.1-2001 permits [EACCES] to be returned for conditions other than those explicitly
 27319 listed.

27320 The O_NOCTTY flag was added to allow applications to avoid unintentionally acquiring a
 27321 controlling terminal as a side effect of opening a terminal file. This volume of
 27322 IEEE Std 1003.1-2001 does not specify how a controlling terminal is acquired, but it allows an
 27323 implementation to provide this on *open()* if the O_NOCTTY flag is not set and other conditions
 27324 specified in the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11, General Terminal
 27325 Interface are met. The O_NOCTTY flag is an effective no-op if the file being opened is not a
 27326 terminal device.

27327 In historical implementations the value of O_RDONLY is zero. Because of that, it is not possible
 27328 to detect the presence of O_RDONLY and another option. Future implementations should
 27329 encode O_RDONLY and O_WRONLY as bit flags so that:

27330 `O_RDONLY | O_WRONLY == O_RDWR`

27331 In general, the *open()* function follows the symbolic link if *path* names a symbolic link. However,
 27332 the *open()* function, when called with O_CREAT and O_EXCL, is required to fail with [EEXIST]
 27333 if *path* names an existing symbolic link, even if the symbolic link refers to a nonexistent file. This
 27334 behavior is required so that privileged applications can create a new file in a known location
 27335 without the possibility that a symbolic link might cause the file to be created in a different
 27336 location.

27337 For example, a privileged application that must create a file with a predictable name in a user-
 27338 writable directory, such as the user's home directory, could be compromised if the user creates a
 27339 symbolic link with that name that refers to a nonexistent file in a system directory. If the user can
 27340 influence the contents of a file, the user could compromise the system by creating a new system
 27341 configuration or spool file that would then be interpreted by the system. The test for a symbolic
 27342 link which refers to a nonexistent file must be atomic with the creation of a new file.

27343 The POSIX.1-1990 standard required that the group ID of a newly created file be set to the group
 27344 ID of its parent directory or to the effective group ID of the creating process. FIPS 151-2 required
 27345 that implementations provide a way to have the group ID be set to the group ID of the
 27346 containing directory, but did not prohibit implementations also supporting a way to set the
 27347 group ID to the effective group ID of the creating process. Conforming applications should not
 27348 assume which group ID will be used. If it matters, an application can use *chown()* to set the
 27349 group ID after the file is created, or determine under what conditions the implementation will
 27350 set the desired group ID.

27351 FUTURE DIRECTIONS

27352 None.

27353 SEE ALSO

27354 *chmod()*, *close()*, *creat()*, *dup()*, *fcntl()*, *lseek()*, *read()*, *umask()*, *unlockpt()*, *write()*, the Base
 27355 Definitions volume of IEEE Std 1003.1-2001, <fcntl.h>, <sys/stat.h>, <sys/types.h>

27356 **CHANGE HISTORY**

27357 First released in Issue 1. Derived from Issue 1 of the SVID.

27358 **Issue 5**

27359 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
27360 Threads Extension.

27361 Large File Summit extensions are added.

27362 **Issue 6**

27363 In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

27364 The following new requirements on POSIX implementations derive from alignment with the
27365 Single UNIX Specification:

27366 • The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was
27367 required for conforming implementations of previous POSIX specifications, it was not
27368 required for UNIX applications.

27369 • In the DESCRIPTION, `O_CREAT` is amended to state that the group ID of the file is set to the
27370 group ID of the file's parent directory or to the effective group ID of the process. This is a
27371 FIPS requirement.

27372 • In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file
27373 description. This change is to support large files.

27374 • In the ERRORS section, the `[Eoverflow]` condition is added. This change is to support
27375 large files.

27376 • The `[ENxio]` mandatory error condition is added.

27377 • The `[EInval]`, `[ENametoolong]`, and `[ETxtbsy]` optional error conditions are added.

27378 The DESCRIPTION and ERRORS sections are updated so that items related to the optional XSI
27379 STREAMS Option Group are marked.

27380 The following changes were made to align with the IEEE P1003.1a draft standard:

27381 • An explanation is added of the effect of the `O_CREAT` and `O_EXCL` flags when the path
27382 refers to a symbolic link.

27383 • The `[Eloop]` optional error condition is added.

27384 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

27385 The DESCRIPTION of `O_EXCL` is updated in response to IEEE PASC Interpretation 1003.1c #48.

27386 NAME

27387 `opendir` — open a directory

27388 SYNOPSIS

27389 `#include <dirent.h>`

27390 `DIR *opendir(const char *dirname);`

27391 DESCRIPTION

27392 The *opendir()* function shall open a directory stream corresponding to the directory named by
 27393 the *dirname* argument. The directory stream is positioned at the first entry. If the type **DIR** is
 27394 implemented using a file descriptor, applications shall only be able to open up to a total of
 27395 {OPEN_MAX} files and directories.

27396 RETURN VALUE

27397 Upon successful completion, *opendir()* shall return a pointer to an object of type **DIR**.
 27398 Otherwise, a null pointer shall be returned and *errno* set to indicate the error.

27399 ERRORS

27400 The *opendir()* function shall fail if:

27401 [EACCES] Search permission is denied for the component of the path prefix of *dirname* or
 27402 read permission is denied for *dirname*.

27403 [ELOOP] A loop exists in symbolic links encountered during resolution of the *dirname*
 27404 argument.

27405 [ENAMETOOLONG] The length of the *dirname* argument exceeds {PATH_MAX} or a pathname
 27406 component is longer than {NAME_MAX}.
 27407

27408 [ENOENT] A component of *dirname* does not name an existing directory or *dirname* is an
 27409 empty string.

27410 [ENOTDIR] A component of *dirname* is not a directory.

27411 The *opendir()* function may fail if:

27412 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 27413 resolution of the *dirname* argument.

27414 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

27415 [ENAMETOOLONG] As a result of encountering a symbolic link in resolution of the *dirname*
 27416 argument, the length of the substituted pathname string exceeded
 27417 {PATH_MAX}.
 27418

27419 [ENFILE] Too many files are currently open in the system.

27420 **EXAMPLES**27421 **Open a Directory Stream**

27422 The following program fragment demonstrates how the *opendir()* function is used.

```

27423 #include <sys/types.h>
27424 #include <dirent.h>
27425 #include <libgen.h>
27426 ...
27427     DIR *dir;
27428     struct dirent *dp;
27429 ...
27430     if ((dir = opendir(".")) == NULL) {
27431         perror ("Cannot open .");
27432         exit (1);
27433     }
27434     while ((dp = readdir (dir)) != NULL) {
27435         ...

```

27436 **APPLICATION USAGE**

27437 The *opendir()* function should be used in conjunction with *readdir()*, *closedir()*, and *rewinddir()* to
 27438 examine the contents of the directory (see the EXAMPLES section in *readdir()*). This method is
 27439 recommended for portability.

27440 **RATIONALE**

27441 Based on historical implementations, the rules about file descriptors apply to directory streams
 27442 as well. However, this volume of IEEE Std 1003.1-2001 does not mandate that the directory
 27443 stream be implemented using file descriptors. The description of *closedir()* clarifies that if a file
 27444 descriptor is used for the directory stream, it is mandatory that *closedir()* deallocate the file
 27445 descriptor. When a file descriptor is used to implement the directory stream, it behaves as if the
 27446 FD_CLOEXEC had been set for the file descriptor.

27447 The directory entries for dot and dot-dot are optional. This volume of IEEE Std 1003.1-2001 does
 27448 not provide a way to test *a priori* for their existence because an application that is portable must
 27449 be written to look for (and usually ignore) those entries. Writing code that presumes that they
 27450 are the first two entries does not always work, as many implementations permit them to be
 27451 other than the first two entries, with a “normal” entry preceding them. There is negligible value
 27452 in providing a way to determine what the implementation does because the code to deal with
 27453 dot and dot-dot must be written in any case and because such a flag would add to the list of
 27454 those flags (which has proven in itself to be objectionable) and might be abused.

27455 Since the structure and buffer allocation, if any, for directory operations are defined by the
 27456 implementation, this volume of IEEE Std 1003.1-2001 imposes no portability requirements for
 27457 erroneous program constructs, erroneous data, or the use of unspecified values such as the use
 27458 or referencing of a *dirp* value or a **dirent** structure value after a directory stream has been closed
 27459 or after a *fork()* or one of the *exec* function calls.

27460 **FUTURE DIRECTIONS**

27461 None.

27462 **SEE ALSO**

27463 *closedir()*, *lstat()*, *readdir()*, *rewinddir()*, *symlink()*, the Base Definitions volume of
 27464 IEEE Std 1003.1-2001, <dirent.h>, <limits.h>, <sys/types.h>

27465 **CHANGE HISTORY**

27466 First released in Issue 2.

27467 **Issue 6**

27468 In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

27469 The following new requirements on POSIX implementations derive from alignment with the
27470 Single UNIX Specification:

27471 • The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was
27472 required for conforming implementations of previous POSIX specifications, it was not
27473 required for UNIX applications.

27474 • The [ELOOP] mandatory error condition is added.

27475 • A second [ENAMETOOLONG] is added as an optional error condition.

27476 The following changes were made to align with the IEEE P1003.1a draft standard:

27477 • The [ELOOP] optional error condition is added.

27478 **NAME**

27479 openlog — open a connection to the logging facility

27480 **SYNOPSIS**

27481 xSI #include <syslog.h>

27482 void openlog(const char *ident, int logopt, int facility);

27483

27484 **DESCRIPTION**27485 Refer to *closelog()*.

27486 **NAME**

27487 optarg, opterr, optind, optopt — options parsing variables

27488 **SYNOPSIS**

27489 #include <unistd.h>

27490 extern char *optarg;

27491 extern int opterr, optind, optopt;

27492 **DESCRIPTION**27493 Refer to *getopt()*.

27494 **NAME**

27495 pathconf — get configurable pathname variables

27496 **SYNOPSIS**

27497 #include <unistd.h>

27498 long pathconf(const char *path, int name);

27499 **DESCRIPTION**27500 Refer to *fpathconf()*.

27501 NAME

27502 pause — suspend the thread until a signal is received

27503 SYNOPSIS

27504 #include <unistd.h>

27505 int pause(void);

27506 DESCRIPTION

27507 The *pause()* function shall suspend the calling thread until delivery of a signal whose action is
27508 either to execute a signal-catching function or to terminate the process.

27509 If the action is to terminate the process, *pause()* shall not return.

27510 If the action is to execute a signal-catching function, *pause()* shall return after the signal-catching
27511 function returns.

27512 RETURN VALUE

27513 Since *pause()* suspends thread execution indefinitely unless interrupted by a signal, there is no
27514 successful completion return value. A value of -1 shall be returned and *errno* set to indicate the
27515 error.

27516 ERRORS

27517 The *pause()* function shall fail if:

27518 [EINTR] A signal is caught by the calling process and control is returned from the
27519 signal-catching function.

27520 EXAMPLES

27521 None.

27522 APPLICATION USAGE

27523 Many common uses of *pause()* have timing windows. The scenario involves checking a
27524 condition related to a signal and, if the signal has not occurred, calling *pause()*. When the signal
27525 occurs between the check and the call to *pause()*, the process often blocks indefinitely. The
27526 *sigprocmask()* and *sigsuspend()* functions can be used to avoid this type of problem.

27527 RATIONALE

27528 None.

27529 FUTURE DIRECTIONS

27530 None.

27531 SEE ALSO

27532 *sigsuspend()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

27533 CHANGE HISTORY

27534 First released in Issue 1. Derived from Issue 1 of the SVID.

27535 Issue 5

27536 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

27537 Issue 6

27538 The APPLICATION USAGE section is added.

27539 **NAME**

27540 pclose — close a pipe stream to or from a process

27541 **SYNOPSIS**27542 **CX** #include <stdio.h>

27543 int pclose(FILE *stream);

27544

27545 **DESCRIPTION**

27546 The *pclose()* function shall close a stream that was opened by *popen()*, wait for the command to
 27547 terminate, and return the termination status of the process that was running the command
 27548 language interpreter. However, if a call caused the termination status to be unavailable to
 27549 *pclose()*, then *pclose()* shall return -1 with *errno* set to [ECHILD] to report this situation. This can
 27550 happen if the application calls one of the following functions:

- 27551 • *wait()*
- 27552 • *waitpid()* with a *pid* argument less than or equal to 0 or equal to the process ID of the
 27553 command line interpreter
- 27554 • Any other function not defined in this volume of IEEE Std 1003.1-2001 that could do one of
 27555 the above

27556 In any case, *pclose()* shall not return before the child process created by *popen()* has terminated.

27557 If the command language interpreter cannot be executed, the child termination status returned
 27558 by *pclose()* shall be as if the command language interpreter terminated using *exit(127)* or
 27559 *_exit(127)*.

27560 The *pclose()* function shall not affect the termination status of any child of the calling process
 27561 other than the one created by *popen()* for the associated stream.

27562 If the argument *stream* to *pclose()* is not a pointer to a stream created by *popen()*, the result of
 27563 *pclose()* is undefined.

27564 **RETURN VALUE**

27565 Upon successful return, *pclose()* shall return the termination status of the command language
 27566 interpreter. Otherwise, *pclose()* shall return -1 and set *errno* to indicate the error.

27567 **ERRORS**

27568 The *pclose()* function shall fail if:

- 27569 [ECHILD] The status of the child process could not be obtained, as described above.

27570 **EXAMPLES**

27571 None.

27572 **APPLICATION USAGE**

27573 None.

27574 **RATIONALE**

27575 There is a requirement that *pclose()* not return before the child process terminates. This is
 27576 intended to disallow implementations that return [EINTR] if a signal is received while waiting.
 27577 If *pclose()* returned before the child terminated, there would be no way for the application to
 27578 discover which child used to be associated with the stream, and it could not do the cleanup
 27579 itself.

27580 If the stream pointed to by *stream* was not created by *popen()*, historical implementations of
 27581 *pclose()* return -1 without setting *errno*. To avoid requiring *pclose()* to set *errno* in this case,
 27582 IEEE Std 1003.1-2001 makes the behavior unspecified. An application should not use *pclose()* to

27583 close any stream that was not created by *popen()*.

27584 Some historical implementations of *pclose()* either block or ignore the signals SIGINT, SIGQUIT,
 27585 and SIGHUP while waiting for the child process to terminate. Since this behavior is not
 27586 described for the *pclose()* function in IEEE Std 1003.1-2001, such implementations are not
 27587 conforming. Also, some historical implementations return [EINTR] if a signal is received, even
 27588 though the child process has not terminated. Such implementations are also considered non-
 27589 conforming.

27590 Consider, for example, an application that uses:

```
27591 popen("command", "r")
```

27592 to start *command*, which is part of the same application. The parent writes a prompt to its
 27593 standard output (presumably the terminal) and then reads from the *popen()*ed stream. The child
 27594 reads the response from the user, does some transformation on the response (pathname
 27595 expansion, perhaps) and writes the result to its standard output. The parent process reads the
 27596 result from the pipe, does something with it, and prints another prompt. The cycle repeats.
 27597 Assuming that both processes do appropriate buffer flushing, this would be expected to work.

27598 To conform to IEEE Std 1003.1-2001, *pclose()* must use *waitpid()*, or some similar function,
 27599 instead of *wait()*.

27600 The code sample below illustrates how the *pclose()* function might be implemented on a system
 27601 conforming to IEEE Std 1003.1-2001.

```
27602 int pclose(FILE *stream)
27603 {
27604     int stat;
27605     pid_t pid;
27606
27607     pid = <pid for process created for stream by popen(>
27608     (void) fclose(stream);
27609     while (waitpid(pid, &stat, 0) == -1) {
27610         if (errno != EINTR){
27611             stat = -1;
27612             break;
27613         }
27614     }
27615     return(stat);
27616 }
```

27616 FUTURE DIRECTIONS

27617 None.

27618 SEE ALSO

27619 *fork()*, *popen()*, *waitpid()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

27620 CHANGE HISTORY

27621 First released in Issue 1. Derived from Issue 1 of the SVID.

27622 **NAME**

27623 perror — write error messages to standard error

27624 **SYNOPSIS**

27625 #include <stdio.h>

27626 void perror(const char *s);

27627 **DESCRIPTION**

27628 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 27629 conflict between the requirements described here and the ISO C standard is unintentional. This
 27630 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

27631 The *perror()* function shall map the error number accessed through the symbol *errno* to a
 27632 language-dependent error message, which shall be written to the standard error stream as
 27633 follows:

- 27634 • First (if *s* is not a null pointer and the character pointed to by *s* is not the null byte), the string
 27635 pointed to by *s* followed by a colon and a <space>.
- 27636 • Then an error message string followed by a <newline>.

27637 The contents of the error message strings shall be the same as those returned by *strerror()* with
 27638 argument *errno*.

27639 CX The *perror()* function shall mark the file associated with the standard error stream as having
 27640 been written (*st_ctime*, *st_mtime* marked for update) at some time between its successful
 27641 completion and *exit()*, *abort()*, or the completion of *fflush()* or *fclose()* on *stderr*.

27642 The *perror()* function shall not change the orientation of the standard error stream.

27643 **RETURN VALUE**27644 The *perror()* function shall not return a value.27645 **ERRORS**

27646 No errors are defined.

27647 **EXAMPLES**27648 **Printing an Error Message for a Function**

27649 The following example replaces *bufptr* with a buffer that is the necessary size. If an error occurs,
 27650 the *perror()* function prints a message and the program exits.

```

27651     #include <stdio.h>
27652     #include <stdlib.h>
27653     ...
27654     char *bufptr;
27655     size_t szbuf;
27656     ...
27657     if ((bufptr = malloc(szbuf)) == NULL) {
27658         perror("malloc"); exit(2);
27659     }
27660     ...
```

27661 **APPLICATION USAGE**

27662 None.

27663 **RATIONALE**

27664 None.

27665 **FUTURE DIRECTIONS**

27666 None.

27667 **SEE ALSO**27668 *strerror()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**stdio.h**>27669 **CHANGE HISTORY**

27670 First released in Issue 1. Derived from Issue 1 of the SVID.

27671 **Issue 5**27672 A paragraph is added to the DESCRIPTION indicating that *perror()* does not change the
27673 orientation of the standard error stream.27674 **Issue 6**

27675 Extensions beyond the ISO C standard are marked.

27676 **NAME**

27677 pipe — create an interprocess channel

27678 **SYNOPSIS**

27679 #include <unistd.h>

27680 int pipe(int *filides*[2]);27681 **DESCRIPTION**

27682 The *pipe()* function shall create a pipe and place two file descriptors, one each into the
 27683 arguments *filides*[0] and *filides*[1], that refer to the open file descriptions for the read and write
 27684 ends of the pipe. Their integer values shall be the two lowest available at the time of the *pipe()*
 27685 call. The O_NONBLOCK and FD_CLOEXEC flags shall be clear on both file descriptors. (The
 27686 *fcntl()* function can be used to set both these flags.)

27687 Data can be written to the file descriptor *filides*[1] and read from the file descriptor *filides*[0]. A
 27688 read on the file descriptor *filides*[0] shall access data written to the file descriptor *filides*[1] on a
 27689 first-in-first-out basis. It is unspecified whether *filides*[0] is also open for writing and whether
 27690 *filides*[1] is also open for reading.

27691 A process has the pipe open for reading (correspondingly writing) if it has a file descriptor open
 27692 that refers to the read end, *filides*[0] (write end, *filides*[1]).

27693 Upon successful completion, *pipe()* shall mark for update the *st_atime*, *st_ctime*, and *st_mtime*
 27694 fields of the pipe.

27695 **RETURN VALUE**

27696 Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned and *errno* set to
 27697 indicate the error.

27698 **ERRORS**27699 The *pipe()* function shall fail if:

27700 [EMFILE] More than {OPEN_MAX} minus two file descriptors are already in use by this
 27701 process.

27702 [ENFILE] The number of simultaneously open files in the system would exceed a
 27703 system-imposed limit.

27704 **EXAMPLES**

27705 None.

27706 **APPLICATION USAGE**

27707 None.

27708 **RATIONALE**

27709 The wording carefully avoids using the verb “to open” in order to avoid any implication of use
 27710 of *open()*; see also *write()*.

27711 **FUTURE DIRECTIONS**

27712 None.

27713 **SEE ALSO**

27714 *fcntl()*, *read()*, *write()*, the Base Definitions volume of IEEE Std 1003.1-2001, <*fcntl.h*>,
 27715 <*unistd.h*>

27716 **CHANGE HISTORY**

27717 First released in Issue 1. Derived from Issue 1 of the SVID.

27718 **Issue 6**

27719 The following new requirements on POSIX implementations derive from alignment with the
27720 Single UNIX Specification:

- 27721 • The DESCRIPTION is updated to indicate that certain dispositions of *fildev[0]* and *fildev[1]*
27722 are unspecified.

27723 NAME

27724 poll — input/output multiplexing

27725 SYNOPSIS

27726 XSI `#include <poll.h>`27727 `int poll(struct pollfd fds[], nfds_t nfds, int timeout);`

27728

27729 DESCRIPTION

27730 The `poll()` function provides applications with a mechanism for multiplexing input/output over
 27731 a set of file descriptors. For each member of the array pointed to by `fds`, `poll()` shall examine the
 27732 given file descriptor for the event(s) specified in `events`. The number of **pollfd** structures in the
 27733 `fds` array is specified by `nfds`. The `poll()` function shall identify those file descriptors on which an
 27734 application can read or write data, or on which certain events have occurred.

27735 The `fds` argument specifies the file descriptors to be examined and the events of interest for each
 27736 file descriptor. It is a pointer to an array with one member for each open file descriptor of
 27737 interest. The array's members are **pollfd** structures within which `fd` specifies an open file
 27738 descriptor and `events` and `revents` are bitmasks constructed by OR'ing a combination of the
 27739 following event flags:

27740	POLLIN	Data other than high-priority data may be read without blocking.
27741 XSR		For STREAMS, this flag is set in <code>revents</code> even if the message is of zero length.
27742		This flag shall be equivalent to <code>POLLRDNORM POLLRDBAND</code> .
27743	POLLRDNORM	Normal data may be read without blocking.
27744 XSR		For STREAMS, data on priority band 0 may be read without blocking. This
27745		flag is set in <code>revents</code> even if the message is of zero length.
27746	POLLRDBAND	Priority data may be read without blocking.
27747 XSR		For STREAMS, data on priority bands greater than 0 may be read without
27748		blocking. This flag is set in <code>revents</code> even if the message is of zero length.
27749	POLLPRI	High-priority data may be read without blocking.
27750 XSR		For STREAMS, this flag is set in <code>revents</code> even if the message is of zero length.
27751	POLLOUT	Normal data may be written without blocking.
27752 XSR		For STREAMS, data on priority band 0 may be written without blocking.
27753	POLLWRNORM	Equivalent to <code>POLLOUT</code> .
27754	POLLWRBAND	Priority data may be written.
27755 XSR		For STREAMS, data on priority bands greater than 0 may be written without
27756		blocking. If any priority band has been written to on this STREAM, this event
27757		only examines bands that have been written to at least once.
27758	POLLERR	An error has occurred on the device or stream. This flag is only valid in the
27759		<code>revents</code> bitmask; it shall be ignored in the <code>events</code> member.
27760	POLLHUP	The device has been disconnected. This event and <code>POLLOUT</code> are mutually-
27761		exclusive; a stream can never be writable if a hangup has occurred. However,
27762		this event and <code>POLLIN</code> , <code>POLLRDNORM</code> , <code>POLLRDBAND</code> , or <code>POLLPRI</code> are not
27763		mutually-exclusive. This flag is only valid in the <code>revents</code> bitmask; it shall be
27764		ignored in the <code>events</code> member.

27765	POLLNVAL	The specified <i>fd</i> value is invalid. This flag is only valid in the <i>revents</i> member;
27766		it shall ignored in the <i>events</i> member.
27767		The significance and semantics of normal, priority, and high-priority data are file and device-
27768		specific.
27769		If the value of <i>fd</i> is less than 0, <i>events</i> shall be ignored, and <i>revents</i> shall be set to 0 in that entry on
27770		return from <i>poll()</i> .
27771		In each pollfd structure, <i>poll()</i> shall clear the <i>revents</i> member, except that where the application
27772		requested a report on a condition by setting one of the bits of <i>events</i> listed above, <i>poll()</i> shall set
27773		the corresponding bit in <i>revents</i> if the requested condition is true. In addition, <i>poll()</i> shall set the
27774		POLLHUP, POLLERR, and POLLNVAL flag in <i>revents</i> if the condition is true, even if the
27775		application did not set the corresponding bit in <i>events</i> .
27776		If none of the defined events have occurred on any selected file descriptor, <i>poll()</i> shall wait at
27777		least <i>timeout</i> milliseconds for an event to occur on any of the selected file descriptors. If the value
27778		of <i>timeout</i> is 0, <i>poll()</i> shall return immediately. If the value of <i>timeout</i> is -1, <i>poll()</i> shall block until
27779		a requested event occurs or until the call is interrupted.
27780		Implementations may place limitations on the granularity of timeout intervals. If the requested
27781		timeout interval requires a finer granularity than the implementation supports, the actual
27782		timeout interval shall be rounded up to the next supported value.
27783		The <i>poll()</i> function shall not be affected by the O_NONBLOCK flag.
27784		The <i>poll()</i> function shall support regular files, terminal and pseudo-terminal devices, FIFOs,
27785 XSR		pipes, sockets and STREAMS-based files. The behavior of <i>poll()</i> on elements of <i>fds</i> that refer to
27786		other types of file is unspecified.
27787		Regular files shall always poll TRUE for reading and writing.
27788		A file descriptor for a socket that is listening for connections shall indicate that it is ready for
27789		reading, once connections are available. A file descriptor for a socket that is connecting
27790		asynchronously shall indicate that it is ready for writing, once a connection has been established.
27791	RETURN VALUE	
27792		Upon successful completion, <i>poll()</i> shall return a non-negative value. A positive value indicates
27793		the total number of file descriptors that have been selected (that is, file descriptors for which the
27794		<i>revents</i> member is non-zero). A value of 0 indicates that the call timed out and no file descriptors
27795		have been selected. Upon failure, <i>poll()</i> shall return -1 and set <i>errno</i> to indicate the error.
27796	ERRORS	
27797		The <i>poll()</i> function shall fail if:
27798	[EAGAIN]	The allocation of internal data structures failed but a subsequent request may
27799		succeed.
27800	[EINTR]	A signal was caught during <i>poll()</i> .
27801 XSR	[EINVAL]	The <i>nfds</i> argument is greater than {OPEN_MAX}, or one of the <i>fd</i> members
27802		refers to a STREAM or multiplexer that is linked (directly or indirectly)
27803		downstream from a multiplexer.

27804 EXAMPLES

27805 **Checking for Events on a Stream**

27806 The following example opens a pair of STREAMS devices and then waits for either one to
 27807 become writable. This example proceeds as follows:

- 27808 1. Sets the *timeout* parameter to 500 milliseconds.
- 27809 2. Opens the STREAMS devices */dev/dev0* and */dev/dev1*, and then polls them, specifying
 27810 POLLOUT and POLLWRBAND as the events of interest.
- 27811 The STREAMS device names */dev/dev0* and */dev/dev1* are only examples of how
 27812 STREAMS devices can be named; STREAMS naming conventions may vary among
 27813 systems conforming to the IEEE Std 1003.1-2001.
- 27814 3. Uses the *ret* variable to determine whether an event has occurred on either of the two
 27815 STREAMS. The *poll()* function is given 500 milliseconds to wait for an event to occur (if it
 27816 has not occurred prior to the *poll()* call).
- 27817 4. Checks the returned value of *ret*. If a positive value is returned, one of the following can
 27818 be done:
 - 27819 a. Priority data can be written to the open STREAM on priority bands greater than 0,
 27820 because the POLLWRBAND event occurred on the open STREAM (*fds[0]* or *fds[1]*).
 - 27821 b. Data can be written to the open STREAM on priority-band 0, because the POLLOUT
 27822 event occurred on the open STREAM (*fds[0]* or *fds[1]*).
- 27823 5. If the returned value is not a positive value, permission to write data to the open STREAM
 27824 (on any priority band) is denied.
- 27825 6. If the POLLHUP event occurs on the open STREAM (*fds[0]* or *fds[1]*), the device on the
 27826 open STREAM has disconnected.

```

27827 #include <stropts.h>
27828 #include <poll.h>
27829 ...
27830 struct pollfd fds[2];
27831 int timeout_msecs = 500;
27832 int ret;
27833 int i;

27834 /* Open STREAMS device. */
27835 fds[0].fd = open("/dev/dev0", ...);
27836 fds[1].fd = open("/dev/dev1", ...);
27837     fds[0].events = POLLOUT | POLLWRBAND;
27838     fds[1].events = POLLOUT | POLLWRBAND;

27839 ret = poll(fds, 2, timeout_msecs);

27840 if (ret > 0) {
27841     /* An event on one of the fds has occurred. */
27842     for (i=0; i<2; i++) {
27843         if (fds[i].revents & POLLWRBAND) {
27844             /* Priority data may be written on device number i. */
27845             ...
27846         }
27847         if (fds[i].revents & POLLOUT) {
```



```

27848             /* Data may be written on device number i. */
27849         ...
27850     }
27851     if (fds[i].revents & POLLHUP) {
27852         /* A hangup has occurred on device number i. */
27853         ...
27854     }
27855 }
27856 }

```

27857 APPLICATION USAGE

27858 None.

27859 RATIONALE

27860 None.

27861 FUTURE DIRECTIONS

27862 None.

27863 SEE ALSO

27864 Section 2.6 (on page 38), *getmsg()*, *putmsg()*, *read()*, *select()*, *write()*, the Base Definitions volume
 27865 of IEEE Std 1003.1-2001, <**poll.h**>, <**stropts.h**>

27866 CHANGE HISTORY

27867 First released in Issue 4, Version 2.

27868 Issue 5

27869 Moved from X/OPEN UNIX extension to BASE.

27870 The description of POLLWRBAND is updated.

27871 Issue 6

27872 Text referring to sockets is added to the DESCRIPTION.

27873 Text relating to the XSI STREAMS Option Group is marked.

27874 The Open Group Corrigendum U055/3 is applied, updating the DESCRIPTION of
 27875 POLLWRBAND.

27876 **NAME**

27877 popen — initiate pipe streams to or from a process

27878 **SYNOPSIS**27879 CX `#include <stdio.h>`27880 `FILE *popen(const char *command, const char *mode);`

27881

27882 **DESCRIPTION**

27883 The *popen()* function shall execute the command specified by the string *command*. It shall create a
 27884 pipe between the calling program and the executed command, and shall return a pointer to a
 27885 stream that can be used to either read from or write to the pipe.

27886 The environment of the executed command shall be as if a child process were created within the
 27887 *popen()* call using the *fork()* function, and the child invoked the *sh* utility using the call:

27888 `execl(shell_path, "sh", "-c", command, (char *)0);`27889 where *shell_path* is an unspecified pathname for the *sh* utility.

27890 The *popen()* function shall ensure that any streams from previous *popen()* calls that remain open
 27891 in the parent process are closed in the new child process.

27892 The *mode* argument to *popen()* is a string that specifies I/O mode:

- 27893 1. If *mode* is *r*, when the child process is started, its file descriptor `STDOUT_FILENO` shall be
 27894 the writable end of the pipe, and the file descriptor *fileno(stream)* in the calling process,
 27895 where *stream* is the stream pointer returned by *popen()*, shall be the readable end of the
 27896 pipe.
- 27897 2. If *mode* is *w*, when the child process is started its file descriptor `STDIN_FILENO` shall be
 27898 the readable end of the pipe, and the file descriptor *fileno(stream)* in the calling process,
 27899 where *stream* is the stream pointer returned by *popen()*, shall be the writable end of the
 27900 pipe.
- 27901 3. If *mode* is any other value, the result is undefined.

27902 After *popen()*, both the parent and the child process shall be capable of executing independently
 27903 before either terminates.

27904 Pipe streams are byte-oriented.

27905 **RETURN VALUE**

27906 Upon successful completion, *popen()* shall return a pointer to an open stream that can be used to
 27907 read or write to the pipe. Otherwise, it shall return a null pointer and may set *errno* to indicate
 27908 the error.

27909 **ERRORS**27910 The *popen()* function may fail if:

27911 [EMFILE] {FOPEN_MAX} or {STREAM_MAX} streams are currently open in the calling
 27912 process.

27913 [EINVAL] The *mode* argument is invalid.27914 The *popen()* function may also set *errno* values as described by *fork()* or *pipe()*.

27915 **EXAMPLES**

27916 None.

27917 **APPLICATION USAGE**

27918 Since open files are shared, a mode *r* command can be used as an input filter and a mode *w*
 27919 command as an output filter.

27920 Buffered reading before opening an input filter may leave the standard input of that filter
 27921 mispositioned. Similar problems with an output filter may be prevented by careful buffer
 27922 flushing; for example, with *fflush()*.

27923 A stream opened by *popen()* should be closed by *pclose()*.

27924 The behavior of *popen()* is specified for values of *mode* of *r* and *w*. Other modes such as *rb* and
 27925 *wb* might be supported by specific implementations, but these would not be portable features.
 27926 Note that historical implementations of *popen()* only check to see if the first character of *mode* is
 27927 *r*. Thus, a *mode* of *robert the robot* would be treated as *mode r*, and a *mode* of *anything else* would be
 27928 treated as *mode w*.

27929 If the application calls *waitpid()* or *waitid()* with a *pid* argument greater than 0, and it still has a
 27930 stream that was called with *popen()* open, it must ensure that *pid* does not refer to the process
 27931 started by *popen()*.

27932 To determine whether or not the environment specified in the Shell and Utilities volume of
 27933 IEEE Std 1003.1-2001 is present, use the function call:

27934 *sysconf*(*_SC_2_VERSION*)

27935 (See *sysconf()*).

27936 **RATIONALE**

27937 The *popen()* function should not be used by programs that have set user (or group) ID privileges.
 27938 The *fork()* and *exec* family of functions (except *execlp()* and *execvp()*), should be used instead.
 27939 This prevents any unforeseen manipulation of the environment of the user that could cause
 27940 execution of commands not anticipated by the calling program.

27941 If the original and *popen()*ed processes both intend to read or write or read and write a common
 27942 file, and either will be using FILE-type C functions (*fread()*, *fwrite()*, and so on), the rules for
 27943 sharing file handles must be observed (see Section 2.5.1 (on page 35)).

27944 **FUTURE DIRECTIONS**

27945 None.

27946 **SEE ALSO**

27947 *pclose()*, *pipe()*, *sysconf()*, *system()*, the Base Definitions volume of IEEE Std 1003.1-2001,
 27948 <*stdio.h*>, the Shell and Utilities volume of IEEE Std 1003.1-2001, *sh*

27949 **CHANGE HISTORY**

27950 First released in Issue 1. Derived from Issue 1 of the SVID.

27951 **Issue 5**

27952 A statement is added to the DESCRIPTION indicating that pipe streams are byte-oriented.

27953 **Issue 6**

27954 The following new requirements on POSIX implementations derive from alignment with the
 27955 Single UNIX Specification:

- 27956 • The optional [EMFILE] error condition is added.

27957 **NAME**27958 posix_fadvise — file advisory information (**ADVANCED REALTIME**)27959 **SYNOPSIS**27960 ADV `#include <fcntl.h>`27961 `int posix_fadvise(int fd, off_t offset, size_t len, int advice);`

27962

27963 **DESCRIPTION**

27964 The *posix_fadvise()* function shall advise the implementation on the expected behavior of the
 27965 application with respect to the data in the file associated with the open file descriptor, *fd*,
 27966 starting at *offset* and continuing for *len* bytes. The specified range need not currently exist in the
 27967 file. If *len* is zero, all data following *offset* is specified. The implementation may use this
 27968 information to optimize handling of the specified data. The *posix_fadvise()* function shall have no
 27969 effect on the semantics of other operations on the specified data, although it may affect the
 27970 performance of other operations.

27971 The advice to be applied to the data is specified by the *advice* parameter and may be one of the
 27972 following values:

27973 **POSIX_FADV_NORMAL**

27974 Specifies that the application has no advice to give on its behavior with respect to the
 27975 specified data. It is the default characteristic if no advice is given for an open file.

27976 **POSIX_FADV_SEQUENTIAL**

27977 Specifies that the application expects to access the specified data sequentially from lower
 27978 offsets to higher offsets.

27979 **POSIX_FADV_RANDOM**

27980 Specifies that the application expects to access the specified data in a random order.

27981 **POSIX_FADV_WILLNEED**

27982 Specifies that the application expects to access the specified data in the near future.

27983 **POSIX_FADV_DONTNEED**

27984 Specifies that the application expects that it will not access the specified data in the near
 27985 future.

27986 **POSIX_FADV_NOREUSE**

27987 Specifies that the application expects to access the specified data once and then not reuse it
 27988 thereafter.

27989 These values are defined in `<fcntl.h>`.27990 **RETURN VALUE**

27991 Upon successful completion, *posix_fadvise()* shall return zero; otherwise, an error number shall
 27992 be returned to indicate the error.

27993 **ERRORS**27994 The *posix_fadvise()* function shall fail if:27995 [EBADF] The *fd* argument is not a valid file descriptor.27996 [EINVAL] The value of *advice* is invalid.27997 [ESPIPE] The *fd* argument is associated with a pipe or FIFO.

27998 EXAMPLES

27999 None.

28000 APPLICATION USAGE

28001 The *posix_fadvise()* function is part of the Advisory Information option and need not be provided
28002 on all implementations.

28003 RATIONALE

28004 None.

28005 FUTURE DIRECTIONS

28006 None.

28007 SEE ALSO

28008 *posix_madvise()*, the Base Definitions volume of IEEE Std 1003.1-2001, <fcntl.h>

28009 CHANGE HISTORY

28010 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

28011 In the SYNOPSIS, the inclusion of <sys/types.h> is no longer required.

28012 **NAME**28013 posix_fallocate — file space control (**ADVANCED REALTIME**)28014 **SYNOPSIS**28015 ADV `#include <fcntl.h>`28016 `int posix_fallocate(int fd, off_t offset, size_t len);`

28017

28018 **DESCRIPTION**

28019 The *posix_fallocate()* function shall ensure that any required storage for regular file data starting
 28020 at *offset* and continuing for *len* bytes is allocated on the file system storage media. If
 28021 *posix_fallocate()* returns successfully, subsequent writes to the specified file data shall not fail
 28022 due to the lack of free space on the file system storage media.

28023 If the *offset+len* is beyond the current file size, then *posix_fallocate()* shall adjust the file size to
 28024 *offset+len*. Otherwise, the file size shall not be changed.

28025 It is implementation-defined whether a previous *posix_fadvise()* call influences allocation
 28026 strategy.

28027 Space allocated via *posix_fallocate()* shall be freed by a successful call to *creat()* or *open()* that
 28028 truncates the size of the file. Space allocated via *posix_fallocate()* may be freed by a successful call
 28029 to *ftruncate()* that reduces the file size to a size smaller than *offset+len*.

28030 **RETURN VALUE**

28031 Upon successful completion, *posix_fallocate()* shall return zero; otherwise, an error number shall
 28032 be returned to indicate the error.

28033 **ERRORS**28034 The *posix_fallocate()* function shall fail if:

- | | | |
|-------|----------|--|
| 28035 | [EBADF] | The <i>fd</i> argument is not a valid file descriptor. |
| 28036 | [EBADF] | The <i>fd</i> argument references a file that was opened without write permission. |
| 28037 | [EFBIG] | The value of <i>offset+len</i> is greater than the maximum file size. |
| 28038 | [EINTR] | A signal was caught during execution. |
| 28039 | [EINVAL] | The <i>len</i> argument was zero or the <i>offset</i> argument was less than zero. |
| 28040 | [EIO] | An I/O error occurred while reading from or writing to a file system. |
| 28041 | [ENODEV] | The <i>fd</i> argument does not refer to a regular file. |
| 28042 | [ENOSPC] | There is insufficient free space remaining on the file system storage media. |
| 28043 | [ESPIPE] | The <i>fd</i> argument is associated with a pipe or FIFO. |

28044 **EXAMPLES**

28045 None.

28046 **APPLICATION USAGE**

28047 The *posix_fallocate()* function is part of the Advisory Information option and need not be
 28048 provided on all implementations.

28049 **RATIONALE**

28050 None.

28051 **FUTURE DIRECTIONS**

28052 None.

28053 **SEE ALSO**28054 *creat()*, *ftruncate()*, *open()*, *unlink()*, the Base Definitions volume of IEEE Std 1003.1-2001,
28055 <fcntl.h>28056 **CHANGE HISTORY**

28057 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

28058 In the SYNOPSIS, the inclusion of <sys/types.h> is no longer required.

28059 **NAME**

28060 `posix_madvise` — memory advisory information and alignment control (**ADVANCED**
 28061 **REALTIME**)

28062 **SYNOPSIS**

28063 ADV `#include <sys/mman.h>`

28064 `int posix_madvise(void *addr, size_t len, int advice);`
 28065

28066 **DESCRIPTION**

28067 MF|SHM The `posix_madvise()` function need only be supported if either the Memory Mapped Files or the
 28068 Shared Memory Objects options are supported.

28069 The `posix_madvise()` function shall advise the implementation on the expected behavior of the
 28070 application with respect to the data in the memory starting at address *addr*, and continuing for
 28071 *len* bytes. The implementation may use this information to optimize handling of the specified
 28072 data. The `posix_madvise()` function shall have no effect on the semantics of access to memory in
 28073 the specified range, although it may affect the performance of access.

28074 The implementation may require that *addr* be a multiple of the page size, which is the value
 28075 returned by `sysconf()` when the name value `_SC_PAGESIZE` is used.

28076 The advice to be applied to the memory range is specified by the *advice* parameter and may be
 28077 one of the following values:

28078 **POSIX_MADV_NORMAL**

28079 Specifies that the application has no advice to give on its behavior with respect to the
 28080 specified range. It is the default characteristic if no advice is given for a range of memory.

28081 **POSIX_MADV_SEQUENTIAL**

28082 Specifies that the application expects to access the specified range sequentially from lower
 28083 addresses to higher addresses.

28084 **POSIX_MADV_RANDOM**

28085 Specifies that the application expects to access the specified range in a random order.

28086 **POSIX_MADV_WILLNEED**

28087 Specifies that the application expects to access the specified range in the near future.

28088 **POSIX_MADV_DONTNEED**

28089 Specifies that the application expects that it will not access the specified range in the near
 28090 future.

28091 These values are defined in the `<sys/mman.h>` header.

28092 **RETURN VALUE**

28093 Upon successful completion, `posix_madvise()` shall return zero; otherwise, an error number shall
 28094 be returned to indicate the error.

28095 **ERRORS**

28096 The `posix_madvise()` function shall fail if:

28097 [EINVAL] The value of *advice* is invalid.

28098 [ENOMEM] Addresses in the range starting at *addr* and continuing for *len* bytes are partly
 28099 or completely outside the range allowed for the address space of the calling
 28100 process.

28101 The `posix_madvise()` function may fail if:

28102 [EINVAL] The value of *addr* is not a multiple of the value returned by *sysconf()* when the
 28103 name value *_SC_PAGESIZE* is used.

28104 [EINVAL] The value of *len* is zero.

28105 EXAMPLES

28106 None.

28107 APPLICATION USAGE

28108 The *posix_madvise()* function is part of the Advisory Information option and need not be
 28109 provided on all implementations.

28110 RATIONALE

28111 None.

28112 FUTURE DIRECTIONS

28113 None.

28114 SEE ALSO

28115 *mmap()*, *posix_fadvise()*, *sysconf()*, the Base Definitions volume of IEEE Std 1003.1-2001,
 28116 <sys/mman.h>

28117 CHANGE HISTORY

28118 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

28119 IEEE PASC Interpretation 1003.1 #102 is applied.

28120 **NAME**

28121 `posix_mem_offset` — find offset and length of a mapped typed memory block (**ADVANCED**
 28122 **REALTIME**)

28123 **SYNOPSIS**

28124 **TYM** `#include <sys/mman.h>`

```
28125        int posix_mem_offset(const void *restrict addr, size_t len,
28126                            off_t *restrict off, size_t *restrict contig_len,
28127                            int *restrict fildes);
28128
```

28129 **DESCRIPTION**

28130 The `posix_mem_offset()` function shall return in the variable pointed to by *off* a value that
 28131 identifies the offset (or location), within a memory object, of the memory block currently
 28132 mapped at *addr*. The function shall return in the variable pointed to by *fildes*, the descriptor used
 28133 (via `mmap()`) to establish the mapping which contains *addr*. If that descriptor was closed since
 28134 the mapping was established, the returned value of *fildes* shall be `-1`. The *len* argument specifies
 28135 the length of the block of the memory object the user wishes the offset for; upon return, the value
 28136 pointed to by *contig_len* shall equal either *len*, or the length of the largest contiguous block of the
 28137 memory object that is currently mapped to the calling process starting at *addr*, whichever is
 28138 smaller.

28139 If the memory object mapped at *addr* is a typed memory object, then if the *off* and *contig_len*
 28140 values obtained by calling `posix_mem_offset()` are used in a call to `mmap()` with a file descriptor
 28141 that refers to the same memory pool as *fildes* (either through the same port or through a different
 28142 port), and that was opened with neither the `POSIX_TYPED_MEM_ALLOCATE` nor the
 28143 `POSIX_TYPED_MEM_ALLOCATE_CONTIG` flag, the typed memory area that is mapped shall
 28144 be exactly the same area that was mapped at *addr* in the address space of the process that called
 28145 `posix_mem_offset()`.

28146 If the memory object specified by *fildes* is not a typed memory object, then the behavior of this
 28147 function is implementation-defined.

28148 **RETURN VALUE**

28149 Upon successful completion, the `posix_mem_offset()` function shall return zero; otherwise, the
 28150 corresponding error status value shall be returned.

28151 **ERRORS**

28152 The `posix_mem_offset()` function shall fail if:

28153 [EACCES] The process has not mapped a memory object supported by this function at
 28154 the given address *addr*.

28155 This function shall not return an error code of [EINTR].

28156 **EXAMPLES**

28157 None.

28158 **APPLICATION USAGE**

28159 None.

28160 **RATIONALE**

28161 None.

28162 **FUTURE DIRECTIONS**

28163 None.

28164 **SEE ALSO**

28165 *mmap()*, *posix_typed_mem_open()*, the Base Definitions volume of IEEE Std 1003.1-2001,
28166 <**sys/mman.h**>

28167 **CHANGE HISTORY**

28168 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

28169 **NAME**28170 posix_memalign — aligned memory allocation (**ADVANCED REALTIME**)28171 **SYNOPSIS**28172 ADV `#include <stdlib.h>`28173 `int posix_memalign(void **memptr, size_t alignment, size_t size);`

28174

28175 **DESCRIPTION**

28176 The *posix_memalign()* function shall allocate *size* bytes aligned on a boundary specified by
 28177 *alignment*, and shall return a pointer to the allocated memory in *memptr*. The value of *alignment*
 28178 shall be a multiple of *sizeof(void *)*, that is also a power of two. Upon successful completion, the
 28179 value pointed to by *memptr* shall be a multiple of *alignment*.

28180 CX The *free()* function shall deallocate memory that has previously been allocated by
 28181 *posix_memalign()*.

28182 **RETURN VALUE**

28183 Upon successful completion, *posix_memalign()* shall return zero; otherwise, an error number
 28184 shall be returned to indicate the error.

28185 **ERRORS**28186 The *posix_memalign()* function shall fail if:

28187 [EINVAL] The value of the alignment parameter is not a power of two multiple of
 28188 *sizeof(void *)*.

28189 [ENOMEM] There is insufficient memory available with the requested alignment.

28190 **EXAMPLES**

28191 None.

28192 **APPLICATION USAGE**

28193 The *posix_memalign()* function is part of the Advisory Information option and need not be
 28194 provided on all implementations.

28195 **RATIONALE**

28196 None.

28197 **FUTURE DIRECTIONS**

28198 None.

28199 **SEE ALSO**28200 *free()*, *malloc()*, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdlib.h>`28201 **CHANGE HISTORY**

28202 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

28203 In the SYNOPSIS, the inclusion of `<sys/types.h>` is no longer required.

28204 NAME

28205 `posix_openpt` — open a pseudo-terminal device

28206 SYNOPSIS

```
28207 xsi    #include <stdlib.h>
28208        #include <fcntl.h>

28209        int posix_openpt(int oflag);
28210
```

28211 DESCRIPTION

28212 The `posix_openpt()` function shall establish a connection between a master device for a pseudo-terminal and a file descriptor. The file descriptor is used by other I/O functions that refer to that pseudo-terminal.

28215 The file status flags and file access modes of the open file description shall be set according to the value of *oflag*.

28217 Values for *oflag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in `<fcntl.h>`:

28219 `O_RDWR` Open for reading and writing.

28220 `O_NOCTTY` If set `posix_openpt()` shall not cause the terminal device to become the controlling terminal for the process.

28222 The behavior of other values for the *oflag* argument is unspecified.

28223 RETURN VALUE

28224 Upon successful completion, the `posix_openpt()` function shall open a master pseudo-terminal device and return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, `-1` shall be returned and *errno* set to indicate the error.

28227 ERRORS

28228 The `posix_openpt()` function shall fail if:

28229 `[EMFILE]` `{OPEN_MAX}` file descriptors are currently open in the calling process.

28230 `[ENFILE]` The maximum allowable number of files is currently open in the system.

28231 The `posix_openpt()` function may fail if:

28232 `[EINVAL]` The value of *oflag* is not valid.

28233 `[EAGAIN]` Out of pseudo-terminal resources.

28234 xsr `[ENOSR]` Out of STREAMS resources.

28235 EXAMPLES

28236 Opening a Pseudo-Terminal and Returning the Name of the Slave Device and a File Descriptor

```
28238        #include <fcntl.h>
28239        #include <stdio.h>

28240        int masterfd, slavefd;
28241        char *slavedevice;

28242        masterfd = posix_openpt(O_RDWR|O_NOCTTY);
28243        if (masterfd == -1
28244            || grantpt(masterfd) == -1
```



```
28245         || unlockpt (masterfd) == -1
28246         || (slavedevice = ptsname (masterfd)) == NULL)
28247         return -1;

28248     printf("slave device is: %s\n", slavedevice);

28249     slavefd = open(slave, O_RDWR|O_NOCTTY);
28250     if (slavefd < 0)
28251         return -1;
```

28252 APPLICATION USAGE

28253 This function is a method for portably obtaining a file descriptor of a master terminal device for
28254 a pseudo-terminal. The *grantpt()* and *ptsname()* functions can be used to manipulate mode and
28255 ownership permissions, and to obtain the name of the slave device, respectively.

28256 RATIONALE

28257 The standard developers considered the matter of adding a special device for cloning master
28258 pseudo-terminals: the */dev/ptmx* device. However, consensus could not be reached, and it was
28259 felt that adding a new function would permit other implementations. The *posix_openpt()*
28260 function is designed to complement the *grantpt()*, *ptsname()*, and *unlockpt()* functions.

28261 On implementations supporting the */dev/ptmx* clone device, opening the master device of a
28262 pseudo-terminal is simply:

```
28263     mfdp = open("/dev/ptmx", oflag );
28264     if (mfdp < 0)
28265         return -1;
```

28266 FUTURE DIRECTIONS

28267 None.

28268 SEE ALSO

28269 *grantpt()*, *open()*, *ptsname()*, *unlockpt()*, the Base Definitions volume of IEEE Std 1003.1-2001,
28270 <fcntl.h>

28271 CHANGE HISTORY

28272 First released in Issue 6.

28273 NAME

28274 `posix_spawn`, `posix_spawnnp` — spawn a process (**ADVANCED REALTIME**)

28275 SYNOPSIS

28276 SPN `#include <spawn.h>`

```
28277        int posix_spawn(pid_t *restrict pid, const char *restrict path,
28278                        const posix_spawn_file_actions_t *file_actions,
28279                        const posix_spawnattr_t *restrict attrp,
28280                        char *const argv[restrict], char *const envp[restrict]);
28281        int posix_spawnnp(pid_t *restrict pid, const char *restrict file,
28282                        const posix_spawn_file_actions_t *file_actions,
28283                        const posix_spawnattr_t *restrict attrp,
28284                        char *const argv[restrict], char *const envp[restrict]);
28285
```

28286 DESCRIPTION

28287 The *posix_spawn()* and *posix_spawnnp()* functions shall create a new process (child process) from
 28288 the specified process image. The new process image shall be constructed from a regular
 28289 executable file called the new process image file.

28290 When a C program is executed as the result of this call, it shall be entered as a C-language
 28291 function call as follows:

```
28292        int main(int argc, char *argv[]);
```

28293 where *argc* is the argument count and *argv* is an array of character pointers to the arguments
 28294 themselves. In addition, the following variable:

```
28295        extern char **environ;
```

28296 shall be initialized as a pointer to an array of character pointers to the environment strings.

28297 The argument *argv* is an array of character pointers to null-terminated strings. The last member
 28298 of this array shall be a null pointer and is not counted in *argc*. These strings constitute the
 28299 argument list available to the new process image. The value in *argv*[0] should point to a filename
 28300 that is associated with the process image being started by the *posix_spawn()* or *posix_spawnnp()*
 28301 function.

28302 The argument *envp* is an array of character pointers to null-terminated strings. These strings
 28303 constitute the environment for the new process image. The environment array is terminated by a
 28304 null pointer.

28305 The number of bytes available for the child process' combined argument and environment lists
 28306 is {ARG_MAX}. The implementation shall specify in the system documentation (see the Base
 28307 Definitions volume of IEEE Std 1003.1-2001, Chapter 2, Conformance) whether any list
 28308 overhead, such as length words, null terminators, pointers, or alignment bytes, is included in
 28309 this total.

28310 The *path* argument to *posix_spawn()* is a pathname that identifies the new process image file to
 28311 execute.

28312 The *file* parameter to *posix_spawnnp()* shall be used to construct a pathname that identifies the
 28313 new process image file. If the *file* parameter contains a slash character, the *file* parameter shall be
 28314 used as the pathname for the new process image file. Otherwise, the path prefix for this file shall
 28315 be obtained by a search of the directories passed as the environment variable *PATH* (see the Base
 28316 Definitions volume of IEEE Std 1003.1-2001, Chapter 8, Environment Variables). If this
 28317 environment variable is not defined, the results of the search are implementation-defined.

28318 If *file_actions* is a null pointer, then file descriptors open in the calling process shall remain open
 28319 in the child process, except for those whose close-on-exec flag FD_CLOEXEC is set (see *fcntl()*).
 28320 For those file descriptors that remain open, all attributes of the corresponding open file
 28321 descriptions, including file locks (see *fcntl()*), shall remain unchanged.

28322 If *file_actions* is not NULL, then the file descriptors open in the child process shall be those open
 28323 in the calling process as modified by the spawn file actions object pointed to by *file_actions* and
 28324 the FD_CLOEXEC flag of each remaining open file descriptor after the spawn file actions have
 28325 been processed. The effective order of processing the spawn file actions shall be:

- 28326 1. The set of open file descriptors for the child process shall initially be the same set as is
 28327 open for the calling process. All attributes of the corresponding open file descriptions,
 28328 including file locks (see *fcntl()*), shall remain unchanged.
- 28329 2. The signal mask, signal default actions, and the effective user and group IDs for the child
 28330 process shall be changed as specified in the attributes object referenced by *attrp*.
- 28331 3. The file actions specified by the spawn file actions object shall be performed in the order in
 28332 which they were added to the spawn file actions object.
- 28333 4. Any file descriptor that has its FD_CLOEXEC flag set (see *fcntl()*) shall be closed.

28334 The **posix_spawnattr_t** spawn attributes object type is defined in **<spawn.h>**. It shall contain at
 28335 least the attributes defined below.

28336 If the POSIX_SPAWN_SETPGROUP flag is set in the *spawn_flags* attribute of the object
 28337 referenced by *attrp*, and the *spawn-pgroup* attribute of the same object is non-zero, then the
 28338 child's process group shall be as specified in the *spawn-pgroup* attribute of the object referenced
 28339 by *attrp*.

28340 As a special case, if the POSIX_SPAWN_SETPGROUP flag is set in the *spawn_flags* attribute of
 28341 the object referenced by *attrp*, and the *spawn-pgroup* attribute of the same object is set to zero,
 28342 then the child shall be in a new process group with a process group ID equal to its process ID.

28343 If the POSIX_SPAWN_SETPGROUP flag is not set in the *spawn_flags* attribute of the object
 28344 referenced by *attrp*, the new child process shall inherit the parent's process group.

28345 PS If the POSIX_SPAWN_SETSCHEDPARAM flag is set in the *spawn_flags* attribute of the object
 28346 referenced by *attrp*, but POSIX_SPAWN_SETSCHEDULER is not set, the new process image
 28347 shall initially have the scheduling policy of the calling process with the scheduling parameters
 28348 specified in the *spawn-schedparam* attribute of the object referenced by *attrp*.

28349 If the POSIX_SPAWN_SETSCHEDULER flag is set in the *spawn_flags* attribute of the object
 28350 referenced by *attrp* (regardless of the setting of the POSIX_SPAWN_SETSCHEDPARAM flag),
 28351 the new process image shall initially have the scheduling policy specified in the *spawn-*
 28352 *schedpolicy* attribute of the object referenced by *attrp* and the scheduling parameters specified in
 28353 the *spawn-schedparam* attribute of the same object.

28354 The POSIX_SPAWN_RESETHIDS flag in the *spawn_flags* attribute of the object referenced by *attrp*
 28355 governs the effective user ID of the child process. If this flag is not set, the child process shall
 28356 inherit the parent process' effective user ID. If this flag is set, the child process' effective user ID
 28357 shall be reset to the parent's real user ID. In either case, if the set-user-ID mode bit of the new
 28358 process image file is set, the effective user ID of the child process shall become that file's owner
 28359 ID before the new process image begins execution.

28360 The POSIX_SPAWN_RESETHIDS flag in the *spawn_flags* attribute of the object referenced by *attrp*
 28361 also governs the effective group ID of the child process. If this flag is not set, the child process
 28362 shall inherit the parent process' effective group ID. If this flag is set, the child process' effective
 28363 group ID shall be reset to the parent's real group ID. In either case, if the set-group-ID mode bit

28364 of the new process image file is set, the effective group ID of the child process shall become that
 28365 file's group ID before the new process image begins execution.

28366 If the POSIX_SPAWN_SETSIGMASK flag is set in the *spawn_flags* attribute of the object
 28367 referenced by *attrp*, the child process shall initially have the signal mask specified in the *spawn-*
 28368 *sigmask* attribute of the object referenced by *attrp*.

28369 If the POSIX_SPAWN_SETSIGDEF flag is set in the *spawn_flags* attribute of the object referenced
 28370 by *attrp*, the signals specified in the *spawn-sigdefault* attribute of the same object shall be set to
 28371 their default actions in the child process. Signals set to the default action in the parent process
 28372 shall be set to the default action in the child process.

28373 Signals set to be caught by the calling process shall be set to the default action in the child
 28374 process.

28375 Except for SIGCHLD, signals set to be ignored by the calling process image shall be set to be
 28376 ignored by the child process, unless otherwise specified by the POSIX_SPAWN_SETSIGDEF flag
 28377 being set in the *spawn_flags* attribute of the object referenced by *attrp* and the signals being
 28378 indicated in the *spawn-sigdefault* attribute of the object referenced by *attrp*.

28379 If the SIGCHLD signal is set to be ignored by the calling process, it is unspecified whether the
 28380 SIGCHLD signal is set to be ignored or to the default action in the child process, unless
 28381 otherwise specified by the POSIX_SPAWN_SETSIGDEF flag being set in the *spawn_flags*
 28382 attribute of the object referenced by *attrp* and the SIGCHLD signal being indicated in the
 28383 *spawn-sigdefault* attribute of the object referenced by *attrp*.

28384 If the value of the *attrp* pointer is NULL, then the default values are used.

28385 All process attributes, other than those influenced by the attributes set in the object referenced
 28386 by *attrp* as specified above or by the file descriptor manipulations specified in *file_actions*, shall
 28387 appear in the new process image as though *fork()* had been called to create a child process and
 28388 then a member of the *exec* family of functions had been called by the child process to execute the
 28389 new process image.

28390 THR It is implementation-defined whether the fork handlers are run when *posix_spawn()* or
 28391 *posix_spawnnp()* is called.

28392 RETURN VALUE

28393 Upon successful completion, *posix_spawn()* and *posix_spawnnp()* shall return the process ID of the
 28394 child process to the parent process, in the variable pointed to by a non-NULL *pid* argument, and
 28395 shall return zero as the function return value. Otherwise, no child process shall be created, the
 28396 value stored into the variable pointed to by a non-NULL *pid* is unspecified, and an error number
 28397 shall be returned as the function return value to indicate the error. If the *pid* argument is a null
 28398 pointer, the process ID of the child is not returned to the caller.

28399 ERRORS

28400 The *posix_spawn()* and *posix_spawnnp()* functions may fail if:

28401 [EINVAL] The value specified by *file_actions* or *attrp* is invalid.

28402 If this error occurs after the calling process successfully returns from the *posix_spawn()* or
 28403 *posix_spawnnp()* function, the child process may exit with exit status 127.

28404 If *posix_spawn()* or *posix_spawnnp()* fail for any of the reasons that would cause *fork()* or one of
 28405 the *exec* family of functions to fail, an error value shall be returned as described by *fork()* and
 28406 *exec*, respectively (or, if the error occurs after the calling process successfully returns, the child
 28407 process shall exit with exit status 127).

28408 If POSIX_SPAWN_SETPGROUP is set in the *spawn-flags* attribute of the object referenced by
 28409 *attrp*, and *posix_spawn()* or *posix_spawnnp()* fails while changing the child's process group, an
 28410 error value shall be returned as described by *setpgid()* (or, if the error occurs after the calling
 28411 process successfully returns, the child process shall exit with exit status 127).

28412 PS If POSIX_SPAWN_SETSCHEDPARAM is set and POSIX_SPAWN_SETSCHEDULER is not set
 28413 in the *spawn-flags* attribute of the object referenced by *attrp*, then if *posix_spawn()* or
 28414 *posix_spawnnp()* fails for any of the reasons that would cause *sched_setparam()* to fail, an error
 28415 value shall be returned as described by *sched_setparam()* (or, if the error occurs after the calling
 28416 process successfully returns, the child process shall exit with exit status 127).

28417 If POSIX_SPAWN_SETSCHEDULER is set in the *spawn-flags* attribute of the object referenced by
 28418 *attrp*, and if *posix_spawn()* or *posix_spawnnp()* fails for any of the reasons that would cause
 28419 *sched_setscheduler()* to fail, an error value shall be returned as described by *sched_setscheduler()*
 28420 (or, if the error occurs after the calling process successfully returns, the child process shall exit
 28421 with exit status 127).

28422 If the *file_actions* argument is not NULL, and specifies any *close*, *dup2*, or *open* actions to be
 28423 performed, and if *posix_spawn()* or *posix_spawnnp()* fails for any of the reasons that would cause
 28424 *close()*, *dup2()*, or *open()* to fail, an error value shall be returned as described by *close()*, *dup2()*,
 28425 and *open()*, respectively (or, if the error occurs after the calling process successfully returns, the
 28426 child process shall exit with exit status 127). An open file action may, by itself, result in any of
 28427 the errors described by *close()* or *dup2()*, in addition to those described by *open()*.

28428 EXAMPLES

28429 None.

28430 APPLICATION USAGE

28431 These functions are part of the Spawn option and need not be provided on all implementations.

28432 RATIONALE

28433 The *posix_spawn()* function and its close relation *posix_spawnnp()* have been introduced to
 28434 overcome the following perceived difficulties with *fork()*: the *fork()* function is difficult or
 28435 impossible to implement without swapping or dynamic address translation.

- 28436 • Swapping is generally too slow for a realtime environment.
- 28437 • Dynamic address translation is not available everywhere that POSIX might be useful.
- 28438 • Processes are too useful to simply option out of POSIX whenever it must run without
- 28439 address translation or other MMU services.

28440 Thus, POSIX needs process creation and file execution primitives that can be efficiently
 28441 implemented without address translation or other MMU services.

28442 The *posix_spawn()* function is implementable as a library routine, but both *posix_spawn()* and
 28443 *posix_spawnnp()* are designed as kernel operations. Also, although they may be an efficient
 28444 replacement for many *fork()/exec* pairs, their goal is to provide useful process creation
 28445 primitives for systems that have difficulty with *fork()*, not to provide drop-in replacements for
 28446 *fork()/exec*.

28447 This view of the role of *posix_spawn()* and *posix_spawnnp()* influenced the design of their API. It
 28448 does not attempt to provide the full functionality of *fork()/exec* in which arbitrary user-specified
 28449 operations of any sort are permitted between the creation of the child process and the execution
 28450 of the new process image; any attempt to reach that level would need to provide a programming
 28451 language as parameters. Instead, *posix_spawn()* and *posix_spawnnp()* are process creation
 28452 primitives like the *Start_Process* and *Start_Process_Search* Ada language bindings package
 28453 *POSIX_Process_Primitives* and also like those in many operating systems that are not UNIX

systems, but with some POSIX-specific additions.

To achieve its coverage goals, *posix_spawn()* and *posix_spawnnp()* have control of six types of inheritance: file descriptors, process group ID, user and group ID, signal mask, scheduling, and whether each signal ignored in the parent will remain ignored in the child, or be reset to its default action in the child.

Control of file descriptors is required to allow an independently written child process image to access data streams opened by and even generated or read by the parent process without being specifically coded to know which parent files and file descriptors are to be used. Control of the process group ID is required to control how the child process' job control relates to that of the parent.

Control of the signal mask and signal defaulting is sufficient to support the implementation of *system()*. Although support for *system()* is not explicitly one of the goals for *posix_spawn()* and *posix_spawnnp()*, it is covered under the "at least 50%" coverage goal.

The intention is that the normal file descriptor inheritance across *fork()*, the subsequent effect of the specified spawn file actions, and the normal file descriptor inheritance across one of the *exec* family of functions should fully specify open file inheritance. The implementation need make no decisions regarding the set of open file descriptors when the child process image begins execution, those decisions having already been made by the caller and expressed as the set of open file descriptors and their *FD_CLOEXEC* flags at the time of the call and the spawn file actions object specified in the call. We have been assured that in cases where the POSIX *Start_Process* Ada primitives have been implemented in a library, this method of controlling file descriptor inheritance may be implemented very easily.

We can identify several problems with *posix_spawn()* and *posix_spawnnp()*, but there does not appear to be a solution that introduces fewer problems. Environment modification for child process attributes not specifiable via the *attrp* or *file_actions* arguments must be done in the parent process, and since the parent generally wants to save its context, it is more costly than similar functionality with *fork()/exec*. It is also complicated to modify the environment of a multi-threaded process temporarily, since all threads must agree when it is safe for the environment to be changed. However, this cost is only borne by those invocations of *posix_spawn()* and *posix_spawnnp()* that use the additional functionality. Since extensive modifications are not the usual case, and are particularly unlikely in time-critical code, keeping much of the environment control out of *posix_spawn()* and *posix_spawnnp()* is appropriate design.

The *posix_spawn()* and *posix_spawnnp()* functions do not have all the power of *fork()/exec*. This is to be expected. The *fork()* function is a wonderfully powerful operation. We do not expect to duplicate its functionality in a simple, fast function with no special hardware requirements. It is worth noting that *posix_spawn()* and *posix_spawnnp()* are very similar to the process creation operations on many operating systems that are not UNIX systems.

Requirements

The requirements for *posix_spawn()* and *posix_spawnnp()* are:

- They must be implementable without an MMU or unusual hardware.
- They must be compatible with existing POSIX standards.

Additional goals are:

- They should be efficiently implementable.
- They should be able to replace at least 50% of typical executions of *fork()*.

- A system with *posix_spawn()* and *posix_spawnnp()* and without *fork()* should be useful, at least for realtime applications.
- A system with *fork()* and the *exec* family should be able to implement *posix_spawn()* and *posix_spawnnp()* as library routines.

Two-Syntax

POSIX *exec* has several calling sequences with approximately the same functionality. These appear to be required for compatibility with existing practice. Since the existing practice for the *posix_spawn*()* functions is otherwise substantially unlike POSIX, we feel that simplicity outweighs compatibility. There are, therefore, only two names for the *posix_spawn*()* functions.

The parameter list does not differ between *posix_spawn()* and *posix_spawnnp()*; *posix_spawnnp()* interprets the second parameter more elaborately than *posix_spawn()*.

Compatibility with POSIX.5 (Ada)

The *Start_Process* and *Start_Process_Search* procedures from the *POSIX_Process_Primitives* package from the Ada language binding to POSIX.1 encapsulate *fork()* and *exec* functionality in a manner similar to that of *posix_spawn()* and *posix_spawnnp()*. Originally, in keeping with our simplicity goal, the standard developers had limited the capabilities of *posix_spawn()* and *posix_spawnnp()* to a subset of the capabilities of *Start_Process* and *Start_Process_Search*; certain non-default capabilities were not supported. However, based on suggestions by the ballot group to improve file descriptor mapping or drop it, and on the advice of an Ada Language Bindings working group member, the standard developers decided that *posix_spawn()* and *posix_spawnnp()* should be sufficiently powerful to implement *Start_Process* and *Start_Process_Search*. The rationale is that if the Ada language binding to such a primitive had already been approved as an IEEE standard, there can be little justification for not approving the functionally-equivalent parts of a C binding. The only three capabilities provided by *posix_spawn()* and *posix_spawnnp()* that are not provided by *Start_Process* and *Start_Process_Search* are optionally specifying the child's process group ID, the set of signals to be reset to default signal handling in the child process, and the child's scheduling policy and parameters.

For the Ada language binding for *Start_Process* to be implemented with *posix_spawn()*, that binding would need to explicitly pass an empty signal mask and the parent's environment to *posix_spawn()* whenever the caller of *Start_Process* allowed these arguments to default, since *posix_spawn()* does not provide such defaults. The ability of *Start_Process* to mask user-specified signals during its execution is functionally unique to the Ada language binding and must be dealt with in the binding separately from the call to *posix_spawn()*.

Process Group

The process group inheritance field can be used to join the child process with an existing process group. By assigning a value of zero to the *spawn-pgroup* attribute of the object referenced by *attrp*, the *setpgid()* mechanism will place the child process in a new process group.

Threads

Without the *posix_spawn()* and *posix_spawnnp()* functions, systems without address translation can still use threads to give an abstraction of concurrency. In many cases, thread creation suffices, but it is not always a good substitute. The *posix_spawn()* and *posix_spawnnp()* functions are considerably “heavier” than thread creation. Processes have several important attributes that threads do not. Even without address translation, a process may have base-and-bound memory protection. Each process has a process environment including security attributes and file capabilities, and powerful scheduling attributes. Processes abstract the behavior of non-uniform-memory-architecture multi-processors better than threads, and they are more convenient to use for activities that are not closely linked.

The *posix_spawn()* and *posix_spawnnp()* functions may not bring support for multiple processes to every configuration. Process creation is not the only piece of operating system support required to support multiple processes. The total cost of support for multiple processes may be quite high in some circumstances. Existing practice shows that support for multiple processes is uncommon and threads are common among “tiny kernels”. There should, therefore, probably continue to be AEPs for operating systems with only one process.

Asynchronous Error Notification

A library implementation of *posix_spawn()* or *posix_spawnnp()* may not be able to detect all possible errors before it forks the child process. IEEE Std 1003.1-2001 provides for an error indication returned from a child process which could not successfully complete the spawn operation via a special exit status which may be detected using the status value returned by *wait()* and *waitpid()*.

The *stat_val* interface and the macros used to interpret it are not well suited to the purpose of returning API errors, but they are the only path available to a library implementation. Thus, an implementation may cause the child process to exit with exit status 127 for any error detected during the spawn process after the *posix_spawn()* or *posix_spawnnp()* function has successfully returned.

The standard developers had proposed using two additional macros to interpret *stat_val*. The first, WIFSPAWNFAIL, would have detected a status that indicated that the child exited because of an error detected during the *posix_spawn()* or *posix_spawnnp()* operations rather than during actual execution of the child process image; the second, WSPAWNERRNO, would have extracted the error value if WIFSPAWNFAIL indicated a failure. Unfortunately, the ballot group strongly opposed this because it would make a library implementation of *posix_spawn()* or *posix_spawnnp()* dependent on kernel modifications to *waitpid()* to be able to embed special information in *stat_val* to indicate a spawn failure.

The 8 bits of child process exit status that are guaranteed by IEEE Std 1003.1-2001 to be accessible to the waiting parent process are insufficient to disambiguate a spawn error from any other kind of error that may be returned by an arbitrary process image. No other bits of the exit status are required to be visible in *stat_val*, so these macros could not be strictly implemented at the library level. Reserving an exit status of 127 for such spawn errors is consistent with the use of this value by *system()* and *popen()* to signal failures in these operations that occur after the function has returned but before a shell is able to execute. The exit status of 127 does not uniquely identify this class of error, nor does it provide any detailed information on the nature of the failure. Note that a kernel implementation of *posix_spawn()* or *posix_spawnnp()* is permitted (and encouraged) to return any possible error as the function value, thus providing more detailed failure information to the parent process.

Thus, no special macros are available to isolate asynchronous *posix_spawn()* or *posix_spawnnp()* errors. Instead, errors detected by the *posix_spawn()* or *posix_spawnnp()* operations in the context

28583 of the child process before the new process image executes are reported by setting the child's
28584 exit status to 127. The calling process may use the WIFEXITED and WEXITSTATUS macros on
28585 the *stat_val* stored by the *wait()* or *waitpid()* functions to detect spawn failures to the extent that
28586 other status values with which the child process image may exit (before the parent can
28587 conclusively determine that the child process image has begun execution) are distinct from exit
28588 status 127.

28589 FUTURE DIRECTIONS

28590 None.

28591 SEE ALSO

28592 *alarm()*, *chmod()*, *close()*, *dup()*, *exec*, *exit()*, *fcntl()*, *fork()*, *kill()*, *open()*,
28593 *posix_spawn_file_actions_addclose()*, *posix_spawn_file_actions_adddup2()*,
28594 *posix_spawn_file_actions_addopen()*, *posix_spawn_file_actions_destroy()*, <REFERENCE
28595 UNDEFINED>(*posix_spawn_file_actions_init()*, *posix_spawnattr_destroy()*, *posix_spawnattr_init()*,
28596 *posix_spawnattr_getsigdefault()*, *posix_spawnattr_getflags()*, *posix_spawnattr_getpgroup()*,
28597 *posix_spawnattr_getschedparam()*, *posix_spawnattr_getschedpolicy()*, *posix_spawnattr_getsigmask()*,
28598 *posix_spawnattr_setsigdefault()*, *posix_spawnattr_setflags()*, *posix_spawnattr_setpgroup()*,
28599 *posix_spawnattr_setschedparam()*, *posix_spawnattr_setschedpolicy()*, *posix_spawnattr_setsigmask()*,
28600 *sched_setparam()*, *sched_setscheduler()*, *setpgid()*, *setuid()*, *stat()*, *times()*, *wait()*, the Base
28601 Definitions volume of IEEE Std 1003.1-2001, <**spawn.h**>

28602 CHANGE HISTORY

28603 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

28604 IEEE PASC Interpretation 1003.1 #103 is applied, noting that the signal default actions are
28605 changed as well as the signal mask in step 2.

28606 IEEE PASC Interpretation 1003.1 #132 is applied.

28607 NAME

28608 `posix_spawn_file_actions_addclose`, `posix_spawn_file_actions_addopen` — add close or open
28609 action to spawn file actions object (**ADVANCED REALTIME**)

28610 SYNOPSIS

```
28611 SPN    #include <spawn.h>

28612        int posix_spawn_file_actions_addclose(posix_spawn_file_actions_t *
28613                                                file_actions, int fildes);
28614        int posix_spawn_file_actions_addopen(posix_spawn_file_actions_t *
28615                                                restrict file_actions, int fildes,
28616                                                const char *restrict path, int oflag, mode_t mode);
28617
```

28618 DESCRIPTION

28619 These functions shall add or delete a close or open action to a spawn file actions object.

28620 A spawn file actions object is of type **posix_spawn_file_actions_t** (defined in `<spawn.h>`) and is
28621 used to specify a series of actions to be performed by a `posix_spawn()` or `posix_spawnnp()`
28622 operation in order to arrive at the set of open file descriptors for the child process given the set of
28623 open file descriptors of the parent. IEEE Std 1003.1-2001 does not define comparison or
28624 assignment operators for the type **posix_spawn_file_actions_t**.

28625 A spawn file actions object, when passed to `posix_spawn()` or `posix_spawnnp()`, shall specify how
28626 the set of open file descriptors in the calling process is transformed into a set of potentially open
28627 file descriptors for the spawned process. This transformation shall be as if the specified sequence
28628 of actions was performed exactly once, in the context of the spawned process (prior to execution
28629 of the new process image), in the order in which the actions were added to the object;
28630 additionally, when the new process image is executed, any file descriptor (from this new set)
28631 which has its `FD_CLOEXEC` flag set shall be closed (see `posix_spawn()`).

28632 The `posix_spawn_file_actions_addclose()` function shall add a *close* action to the object referenced
28633 by *file_actions* that shall cause the file descriptor *fildes* to be closed (as if `close(fildes)` had been
28634 called) when a new process is spawned using this file actions object.

28635 The `posix_spawn_file_actions_addopen()` function shall add an *open* action to the object referenced
28636 by *file_actions* that shall cause the file named by *path* to be opened (as if `open(path, oflag, mode)`
28637 had been called, and the returned file descriptor, if not *fildes*, had been changed to *fildes*) when a
28638 new process is spawned using this file actions object. If *fildes* was already an open file descriptor,
28639 it shall be closed before the new file is opened.

28640 The string described by *path* shall be copied by the `posix_spawn_file_actions_addopen()` function.

28641 RETURN VALUE

28642 Upon successful completion, these functions shall return zero; otherwise, an error number shall
28643 be returned to indicate the error.

28644 ERRORS

28645 These functions shall fail if:

28646 [EBADF] The value specified by *fildes* is negative or greater than or equal to
28647 {OPEN_MAX}.

28648 These functions may fail if:

28649 [EINVAL] The value specified by *file_actions* is invalid.

28650 [ENOMEM] Insufficient memory exists to add to the spawn file actions object.

28651 It shall not be considered an error for the *files* argument passed to these functions to specify a
 28652 file descriptor for which the specified operation could not be performed at the time of the call.
 28653 Any such error will be detected when the associated file actions object is later used during a
 28654 *posix_spawn()* or *posix_spawnnp()* operation.

28655 EXAMPLES

28656 None.

28657 APPLICATION USAGE

28658 These functions are part of the Spawn option and need not be provided on all implementations.

28659 RATIONALE

28660 A spawn file actions object may be initialized to contain an ordered sequence of *close()*, *dup2()*,
 28661 and *open()* operations to be used by *posix_spawn()* or *posix_spawnnp()* to arrive at the set of open
 28662 file descriptors inherited by the spawned process from the set of open file descriptors in the
 28663 parent at the time of the *posix_spawn()* or *posix_spawnnp()* call. It had been suggested that the
 28664 *close()* and *dup2()* operations alone are sufficient to rearrange file descriptors, and that files
 28665 which need to be opened for use by the spawned process can be handled either by having the
 28666 calling process open them before the *posix_spawn()* or *posix_spawnnp()* call (and close them after),
 28667 or by passing filenames to the spawned process (in *argv*) so that it may open them itself. The
 28668 standard developers recommend that applications use one of these two methods when practical,
 28669 since detailed error status on a failed open operation is always available to the application this
 28670 way. However, the standard developers feel that allowing a spawn file actions object to specify
 28671 open operations is still appropriate because:

- 28672 1. It is consistent with equivalent POSIX.5 (Ada) functionality.
- 28673 2. It supports the I/O redirection paradigm commonly employed by POSIX programs
 28674 designed to be invoked from a shell. When such a program is the child process, it may not
 28675 be designed to open files on its own.
- 28676 3. It allows file opens that might otherwise fail or violate file ownership/access rights if
 28677 executed by the parent process.

28678 Regarding 2. above, note that the spawn open file action provides to *posix_spawn()* and
 28679 *posix_spawnnp()* the same capability that the shell redirection operators provide to *system()*, only
 28680 without the intervening execution of a shell; for example:

```
28681 system ("myprog <file1 3<file2");
```

28682 Regarding 3. above, note that if the calling process needs to open one or more files for access by
 28683 the spawned process, but has insufficient spare file descriptors, then the open action is necessary
 28684 to allow the *open()* to occur in the context of the child process after other file descriptors have
 28685 been closed (that must remain open in the parent).

28686 Additionally, if a parent is executed from a file having a “set-user-id” mode bit set and the
 28687 POSIX_SPAWN_RESETPID flag is set in the spawn attributes, a file created within the parent
 28688 process will (possibly incorrectly) have the parent’s effective user ID as its owner, whereas a file
 28689 created via an *open()* action during *posix_spawn()* or *posix_spawnnp()* will have the parent’s real
 28690 ID as its owner; and an open by the parent process may successfully open a file to which the real
 28691 user should not have access or fail to open a file to which the real user should have access.

File Descriptor Mapping

The standard developers had originally proposed using an array which specified the mapping of child file descriptors back to those of the parent. It was pointed out by the ballot group that it is not possible to reshuffle file descriptors arbitrarily in a library implementation of *posix_spawn()* or *posix_spawnnp()* without provision for one or more spare file descriptor entries (which simply may not be available). Such an array requires that an implementation develop a complex strategy to achieve the desired mapping without inadvertently closing the wrong file descriptor at the wrong time.

It was noted by a member of the Ada Language Bindings working group that the approved Ada Language *Start_Process* family of POSIX process primitives use a caller-specified set of file actions to alter the normal *fork()/exec* semantics for inheritance of file descriptors in a very flexible way, yet no such problems exist because the burden of determining how to achieve the final file descriptor mapping is completely on the application. Furthermore, although the file actions interface appears frightening at first glance, it is actually quite simple to implement in either a library or the kernel.

FUTURE DIRECTIONS

None.

SEE ALSO

close(), *dup()*, *open()*, *posix_spawn()*, *posix_spawn_file_actions_adddup2()*, *posix_spawn_file_actions_destroy()*, *posix_spawnnp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <spawn.h>

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

IEEE PASC Interpretation 1003.1 #105 is applied, adding a note to the DESCRIPTION that the string pointed to by *path* is copied by the *posix_spawn_file_actions_addopen()* function.

28717 **NAME**

28718 `posix_spawn_file_actions_adddup2` — add `dup2` action to spawn file actions object
 28719 (**ADVANCED REALTIME**)

28720 **SYNOPSIS**

28721 SPN `#include <spawn.h>`

28722 `int posix_spawn_file_actions_adddup2(posix_spawn_file_actions_t *`
 28723 `file_actions, int fildes, int newfildes);`
 28724

28725 **DESCRIPTION**

28726 The `posix_spawn_file_actions_adddup2()` function shall add a `dup2()` action to the object
 28727 referenced by `file_actions` that shall cause the file descriptor `fildes` to be duplicated as `newfildes` (as
 28728 if `dup2(fildes, newfildes)` had been called) when a new process is spawned using this file actions
 28729 object.

28730 A spawn file actions object is as defined in `posix_spawn_file_actions_addclose()`.

28731 **RETURN VALUE**

28732 Upon successful completion, the `posix_spawn_file_actions_adddup2()` function shall return zero;
 28733 otherwise, an error number shall be returned to indicate the error.

28734 **ERRORS**

28735 The `posix_spawn_file_actions_adddup2()` function shall fail if:

28736 [EBADF] The value specified by `fildes` or `newfildes` is negative or greater than or equal to
 28737 {OPEN_MAX}.

28738 [ENOMEM] Insufficient memory exists to add to the spawn file actions object.

28739 The `posix_spawn_file_actions_adddup2()` function may fail if:

28740 [EINVAL] The value specified by `file_actions` is invalid.

28741 It shall not be considered an error for the `fildes` argument passed to the
 28742 `posix_spawn_file_actions_adddup2()` function to specify a file descriptor for which the specified
 28743 operation could not be performed at the time of the call. Any such error will be detected when
 28744 the associated file actions object is later used during a `posix_spawn()` or `posix_spawnnp()`
 28745 operation.

28746 **EXAMPLES**

28747 None.

28748 **APPLICATION USAGE**

28749 The `posix_spawn_file_actions_adddup2()` function is part of the Spawn option and need not be
 28750 provided on all implementations.

28751 **RATIONALE**

28752 Refer to the RATIONALE in `posix_spawn_file_actions_addclose()`.

28753 **FUTURE DIRECTIONS**

28754 None.

28755 **SEE ALSO**

28756 `dup()`, `posix_spawn()`, `posix_spawn_file_actions_addclose()`, `posix_spawn_file_actions_destroy()`,
 28757 `posix_spawnnp()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<spawn.h>`

28758 CHANGE HISTORY

28759 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

28760 IEEE PASC Interpretation 1003.1 #104 is applied, noting that the [EBADF] error can apply to the
28761 *newfildes* argument in addition to *fildes*.

28762 **NAME**

28763 posix_spawn_file_actions_addopen — add open action to spawn file actions object
28764 (ADVANCED REALTIME)

28765 **SYNOPSIS**

28766 SPN #include <spawn.h>

```
28767       int posix_spawn_file_actions_addopen(posix_spawn_file_actions_t *  
28768       restrict file_actions, int fildes,  
28769       const char *restrict path, int oflag, mode_t mode);  
28770
```

28771 **DESCRIPTION**

28772 Refer to *posix_spawn_file_actions_addclose()*.

28773 NAME

28774 posix_spawn_file_actions_destroy, posix_spawn_file_actions_init — destroy and initialize
28775 spawn file actions object (**ADVANCED REALTIME**)

28776 SYNOPSIS

28777 SPN `#include <spawn.h>`

```
28778 int posix_spawn_file_actions_destroy(posix_spawn_file_actions_t *
28779     file_actions);
28780 int posix_spawn_file_actions_init(posix_spawn_file_actions_t *
28781     file_actions);
28782
```

28783 DESCRIPTION

28784 The *posix_spawn_file_actions_destroy()* function shall destroy the object referenced by *file_actions*;
28785 the object becomes, in effect, uninitialized. An implementation may cause
28786 *posix_spawn_file_actions_destroy()* to set the object referenced by *file_actions* to an invalid value. A
28787 destroyed spawn file actions object can be reinitialized using *posix_spawn_file_actions_init()*; the
28788 results of otherwise referencing the object after it has been destroyed are undefined.

28789 The *posix_spawn_file_actions_init()* function shall initialize the object referenced by *file_actions* to
28790 contain no file actions for *posix_spawn()* or *posix_spawnnp()* to perform.

28791 A spawn file actions object is as defined in *posix_spawn_file_actions_addclose()*.

28792 The effect of initializing an already initialized spawn file actions object is undefined.

28793 RETURN VALUE

28794 Upon successful completion, these functions shall return zero; otherwise, an error number shall
28795 be returned to indicate the error.

28796 ERRORS

28797 The *posix_spawn_file_actions_init()* function shall fail if:

28798 [ENOMEM] Insufficient memory exists to initialize the spawn file actions object.

28799 The *posix_spawn_file_actions_destroy()* function may fail if:

28800 [EINVAL] The value specified by *file_actions* is invalid.

28801 EXAMPLES

28802 None.

28803 APPLICATION USAGE

28804 These functions are part of the Spawn option and need not be provided on all implementations.

28805 RATIONALE

28806 Refer to the RATIONALE in *posix_spawn_file_actions_addclose()*.

28807 FUTURE DIRECTIONS

28808 None.

28809 SEE ALSO

28810 *posix_spawn()*, *posix_spawnnp()*, the Base Definitions volume of IEEE Std 1003.1-2001, `<spawn.h>`

28811 CHANGE HISTORY

28812 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

28813 In the SYNOPSIS, the inclusion of `<sys/types.h>` is no longer required.

28814 NAME

28815 `posix_spawnattr_destroy`, `posix_spawnattr_init` — destroy and initialize spawn attributes object
 28816 (ADVANCED REALTIME)

28817 SYNOPSIS

28818 SPN `#include <spawn.h>`

28819 `int posix_spawnattr_destroy(posix_spawnattr_t *attr);`

28820 `int posix_spawnattr_init(posix_spawnattr_t *attr);`

28821

28822 DESCRIPTION

28823 The `posix_spawnattr_destroy()` function shall destroy a spawn attributes object. A destroyed `attr`
 28824 attributes object can be reinitialized using `posix_spawnattr_init()`; the results of otherwise
 28825 referencing the object after it has been destroyed are undefined. An implementation may cause
 28826 `posix_spawnattr_destroy()` to set the object referenced by `attr` to an invalid value.

28827 The `posix_spawnattr_init()` function shall initialize a spawn attributes object `attr` with the default
 28828 value for all of the individual attributes used by the implementation. Results are undefined if
 28829 `posix_spawnattr_init()` is called specifying an already initialized `attr` attributes object.

28830 A spawn attributes object is of type **posix_spawnattr_t** (defined in `<spawn.h>`) and is used to
 28831 specify the inheritance of process attributes across a spawn operation. IEEE Std 1003.1-2001 does
 28832 not define comparison or assignment operators for the type **posix_spawnattr_t**.

28833 Each implementation shall document the individual attributes it uses and their default values
 28834 unless these values are defined by IEEE Std 1003.1-2001. Attributes not defined by
 28835 IEEE Std 1003.1-2001, their default values, and the names of the associated functions to get and
 28836 set those attribute values are implementation-defined.

28837 The resulting spawn attributes object (possibly modified by setting individual attribute values),
 28838 is used to modify the behavior of `posix_spawn()` or `posix_spawnnp()`. After a spawn attributes
 28839 object has been used to spawn a process by a call to a `posix_spawn()` or `posix_spawnnp()`, any
 28840 function affecting the attributes object (including destruction) shall not affect any process that
 28841 has been spawned in this way.

28842 RETURN VALUE

28843 Upon successful completion, `posix_spawnattr_destroy()` and `posix_spawnattr_init()` shall return
 28844 zero; otherwise, an error number shall be returned to indicate the error.

28845 ERRORS

28846 The `posix_spawnattr_init()` function shall fail if:

28847 [ENOMEM] Insufficient memory exists to initialize the spawn attributes object.

28848 The `posix_spawnattr_destroy()` function may fail if:

28849 [EINVAL] The value specified by `attr` is invalid.

28850 EXAMPLES

28851 None.

28852 APPLICATION USAGE

28853 These functions are part of the Spawn option and need not be provided on all implementations.

28854 RATIONALE

28855 The original spawn interface proposed in IEEE Std 1003.1-2001 defined the attributes that specify
 28856 the inheritance of process attributes across a spawn operation as a structure. In order to be able
 28857 to separate optional individual attributes under their appropriate options (that is, the `spawn-`
 28858 `schedparam` and `spawn-schedpolicy` attributes depending upon the Process Scheduling option), and

28859 also for extensibility and consistency with the newer POSIX interfaces, the attributes interface
28860 has been changed to an opaque data type. This interface now consists of the type
28861 **posix_spawnattr_t**, representing a spawn attributes object, together with associated functions to
28862 initialize or destroy the attributes object, and to set or get each individual attribute. Although the
28863 new object-oriented interface is more verbose than the original structure, it is simple to use,
28864 more extensible, and easy to implement.

28865 FUTURE DIRECTIONS

28866 None.

28867 SEE ALSO

28868 *posix_spawn()*, *posix_spawnattr_getsigdefault()*, *posix_spawnattr_getflags()*,
28869 *posix_spawnattr_getpgroup()*, *posix_spawnattr_getschedparam()*, *posix_spawnattr_getschedpolicy()*,
28870 *posix_spawnattr_getsigmask()*, *posix_spawnattr_setsigdefault()*, *posix_spawnattr_setflags()*,
28871 *posix_spawnattr_setpgroup()*, *posix_spawnattr_setsigmask()*, *posix_spawnattr_setschedpolicy()*,
28872 *posix_spawnattr_setschedparam()*, *posix_spawn()*, the Base Definitions volume of
28873 IEEE Std 1003.1-2001, <spawn.h>

28874 CHANGE HISTORY

28875 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

28876 IEEE PASC Interpretation 1003.1 #106 is applied, noting that the effect of initializing an already
28877 initialized spawn attributes option is undefined.

28878 **NAME**

28879 `posix_spawnattr_getflags`, `posix_spawnattr_setflags` — get and set the spawn-flags attribute of a
 28880 spawn attributes object (**ADVANCED REALTIME**)

28881 **SYNOPSIS**

```
28882 SPN    #include <spawn.h>

28883        int posix_spawnattr_getflags(const posix_spawnattr_t *restrict attr,
28884                                    short *restrict flags);
28885        int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags);
28886
```

28887 **DESCRIPTION**

28888 The `posix_spawnattr_getflags()` function shall obtain the value of the *spawn-flags* attribute from
 28889 the attributes object referenced by *attr*.

28890 The `posix_spawnattr_setflags()` function shall set the *spawn-flags* attribute in an initialized
 28891 attributes object referenced by *attr*.

28892 The *spawn-flags* attribute is used to indicate which process attributes are to be changed in the
 28893 new process image when invoking `posix_spawn()` or `posix_spawnnp()`. It is the bitwise-inclusive
 28894 OR of zero or more of the following flags:

```
28895        POSIX_SPAWN_RESETPIDS
28896        POSIX_SPAWN_SETPGROUP
28897        POSIX_SPAWN_SETSIGDEF
28898        POSIX_SPAWN_SETSIGMASK
28899 PS     POSIX_SPAWN_SETSCHEDPARAM
28900        POSIX_SPAWN_SETSCHEDULER
28901
```

28902 These flags are defined in `<spawn.h>`. The default value of this attribute shall be as if no flags
 28903 were set.

28904 **RETURN VALUE**

28905 Upon successful completion, `posix_spawnattr_getflags()` shall return zero and store the value of
 28906 the *spawn-flags* attribute of *attr* into the object referenced by the *flags* parameter; otherwise, an
 28907 error number shall be returned to indicate the error.

28908 Upon successful completion, `posix_spawnattr_setflags()` shall return zero; otherwise, an error
 28909 number shall be returned to indicate the error.

28910 **ERRORS**

28911 These functions may fail if:

28912 [EINVAL] The value specified by *attr* is invalid.

28913 The `posix_spawnattr_setflags()` function may fail if:

28914 [EINVAL] The value of the attribute being set is not valid.

28915 EXAMPLES

28916 None.

28917 APPLICATION USAGE

28918 These functions are part of the Spawn option and need not be provided on all implementations.

28919 RATIONALE

28920 None.

28921 FUTURE DIRECTIONS

28922 None.

28923 SEE ALSO

28924 *posix_spawn()*, *posix_spawnattr_destroy()*, *posix_spawnattr_init()*, *posix_spawnattr_getsigdefault()*,
28925 *posix_spawnattr_getpgroup()*, *posix_spawnattr_getschedparam()*, *posix_spawnattr_getschedpolicy()*,
28926 *posix_spawnattr_getsigmask()*, *posix_spawnattr_setsigdefault()*, *posix_spawnattr_setpgroup()*,
28927 *posix_spawnattr_setschedparam()*, *posix_spawnattr_setschedpolicy()*, *posix_spawnattr_setsigmask()*,
28928 *posix_spawnnp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**spawn.h**>

28929 CHANGE HISTORY

28930 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

28931 **NAME**

28932 `posix_spawnattr_getpgroup`, `posix_spawnattr_setpgroup` — get and set the spawn-pgroup
 28933 attribute of a spawn attributes object (**ADVANCED REALTIME**)

28934 **SYNOPSIS**

```
28935 SPN    #include <spawn.h>

28936        int posix_spawnattr_getpgroup(const posix_spawnattr_t *restrict attr,
28937                                      pid_t *restrict pgroup);
28938        int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup);
28939
```

28940 **DESCRIPTION**

28941 The `posix_spawnattr_getpgroup()` function shall obtain the value of the *spawn-pgroup* attribute
 28942 from the attributes object referenced by *attr*.

28943 The `posix_spawnattr_setpgroup()` function shall set the *spawn-pgroup* attribute in an initialized
 28944 attributes object referenced by *attr*.

28945 The *spawn-pgroup* attribute represents the process group to be joined by the new process image
 28946 in a spawn operation (if `POSIX_SPAWN_SETPGROUP` is set in the *spawn-flags* attribute). The
 28947 default value of this attribute shall be zero.

28948 **RETURN VALUE**

28949 Upon successful completion, `posix_spawnattr_getpgroup()` shall return zero and store the value of
 28950 the *spawn-pgroup* attribute of *attr* into the object referenced by the *pgroup* parameter; otherwise,
 28951 an error number shall be returned to indicate the error.

28952 Upon successful completion, `posix_spawnattr_setpgroup()` shall return zero; otherwise, an error
 28953 number shall be returned to indicate the error.

28954 **ERRORS**

28955 These functions may fail if:

28956 [EINVAL] The value specified by *attr* is invalid.

28957 The `posix_spawnattr_setpgroup()` function may fail if:

28958 [EINVAL] The value of the attribute being set is not valid.

28959 **EXAMPLES**

28960 None.

28961 **APPLICATION USAGE**

28962 These functions are part of the Spawn option and need not be provided on all implementations.

28963 **RATIONALE**

28964 None.

28965 **FUTURE DIRECTIONS**

28966 None.

28967 **SEE ALSO**

28968 `posix_spawn()`, `posix_spawnattr_destroy()`, `posix_spawnattr_init()`, `posix_spawnattr_getsigdefault()`,
 28969 `posix_spawnattr_getflags()`, `posix_spawnattr_getschedparam()`, `posix_spawnattr_getschedpolicy()`,
 28970 `posix_spawnattr_getsigmask()`, `posix_spawnattr_setsigdefault()`, `posix_spawnattr_setflags()`,
 28971 `posix_spawnattr_setschedparam()`, `posix_spawnattr_setschedpolicy()`, `posix_spawnattr_setsigmask()`,
 28972 `posix_spawnnp()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<spawn.h>`

28973 CHANGE HISTORY

28974 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

28975 **NAME**

28976 `posix_spawnattr_getschedparam`, `posix_spawnattr_setschedparam` — get and set the spawn-
 28977 `schedparam` attribute of a spawn attributes object (**ADVANCED REALTIME**)

28978 **SYNOPSIS**

28979 SPN PS `#include <spawn.h>`

28980 `#include <sched.h>`

```
28981 int posix_spawnattr_getschedparam(const posix_spawnattr_t *
28982     restrict attr, struct sched_param *restrict schedparam);
28983 int posix_spawnattr_setschedparam(posix_spawnattr_t *restrict attr,
28984     const struct sched_param *restrict schedparam);
28985
```

28986 **DESCRIPTION**

28987 The `posix_spawnattr_getschedparam()` function shall obtain the value of the *spawn-schedparam*
 28988 attribute from the attributes object referenced by *attr*.

28989 The `posix_spawnattr_setschedparam()` function shall set the *spawn-schedparam* attribute in an
 28990 initialized attributes object referenced by *attr*.

28991 The *spawn-schedparam* attribute represents the scheduling parameters to be assigned to the new
 28992 process image in a spawn operation (if `POSIX_SPAWN_SETSCHEDULER` or
 28993 `POSIX_SPAWN_SETSCHEDPARAM` is set in the *spawn-flags* attribute). The default value of this
 28994 attribute is unspecified.

28995 **RETURN VALUE**

28996 Upon successful completion, `posix_spawnattr_getschedparam()` shall return zero and store the
 28997 value of the *spawn-schedparam* attribute of *attr* into the object referenced by the *schedparam*
 28998 parameter; otherwise, an error number shall be returned to indicate the error.

28999 Upon successful completion, `posix_spawnattr_setschedparam()` shall return zero; otherwise, an
 29000 error number shall be returned to indicate the error.

29001 **ERRORS**

29002 These functions may fail if:

29003 [EINVAL] The value specified by *attr* is invalid.

29004 The `posix_spawnattr_setschedparam()` function may fail if:

29005 [EINVAL] The value of the attribute being set is not valid.

29006 **EXAMPLES**

29007 None.

29008 **APPLICATION USAGE**

29009 These functions are part of the Spawn and Process Scheduling options and need not be provided
 29010 on all implementations.

29011 **RATIONALE**

29012 None.

29013 **FUTURE DIRECTIONS**

29014 None.

29015 **SEE ALSO**

29016 `posix_spawn()`, `posix_spawnattr_destroy()`, `posix_spawnattr_init()`, `posix_spawnattr_getsigdefault()`,
 29017 `posix_spawnattr_getflags()`, `posix_spawnattr_getpgroup()`, `posix_spawnattr_getschedpolicy()`,
 29018 `posix_spawnattr_getsigmask()`, `posix_spawnattr_setsigdefault()`, `posix_spawnattr_setflags()`,

29019 *posix_spawnattr_setpgroup()*, *posix_spawnattr_setschedpolicy()*, *posix_spawnattr_setsigmask()*,
29020 *posix_spawnnp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**sched.h**>, <**spawn.h**>

29021 CHANGE HISTORY

29022 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29023 NAME

29024 `posix_spawnattr_getschedpolicy`, `posix_spawnattr_setschedpolicy` — get and set the spawn-
 29025 `schedpolicy` attribute of a spawn attributes object (**ADVANCED REALTIME**)

29026 SYNOPSIS

29027 SPN PS `#include <spawn.h>`

29028 `#include <sched.h>`

```
29029 int posix_spawnattr_getschedpolicy(const posix_spawnattr_t *
29030     restrict attr, int *restrict schedpolicy);
29031 int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr,
29032     int schedpolicy);
29033
```

29034 DESCRIPTION

29035 The `posix_spawnattr_getschedpolicy()` function shall obtain the value of the *spawn-schedpolicy*
 29036 attribute from the attributes object referenced by *attr*.

29037 The `posix_spawnattr_setschedpolicy()` function shall set the *spawn-schedpolicy* attribute in an
 29038 initialized attributes object referenced by *attr*.

29039 The *spawn-schedpolicy* attribute represents the scheduling policy to be assigned to the new
 29040 process image in a spawn operation (if `POSIX_SPAWN_SETSCHEDULER` is set in the *spawn-*
 29041 *flags* attribute). The default value of this attribute is unspecified.

29042 RETURN VALUE

29043 Upon successful completion, `posix_spawnattr_getschedpolicy()` shall return zero and store the
 29044 value of the *spawn-schedpolicy* attribute of *attr* into the object referenced by the *schedpolicy*
 29045 parameter; otherwise, an error number shall be returned to indicate the error.

29046 Upon successful completion, `posix_spawnattr_setschedpolicy()` shall return zero; otherwise, an
 29047 error number shall be returned to indicate the error.

29048 ERRORS

29049 These functions may fail if:

29050 [EINVAL] The value specified by *attr* is invalid.

29051 The `posix_spawnattr_setschedpolicy()` function may fail if:

29052 [EINVAL] The value of the attribute being set is not valid.

29053 EXAMPLES

29054 None.

29055 APPLICATION USAGE

29056 These functions are part of the Spawn and Process Scheduling options and need not be provided
 29057 on all implementations.

29058 RATIONALE

29059 None.

29060 FUTURE DIRECTIONS

29061 None.

29062 SEE ALSO

29063 `posix_spawn()`, `posix_spawnattr_destroy()`, `posix_spawnattr_init()`, `posix_spawnattr_getsigdefault()`,
 29064 `posix_spawnattr_getflags()`, `posix_spawnattr_getpgroup()`, `posix_spawnattr_getschedparam()`,
 29065 `posix_spawnattr_getsigmask()`, `posix_spawnattr_setsigdefault()`, `posix_spawnattr_setflags()`,
 29066 `posix_spawnattr_setpgroup()`, `posix_spawnattr_setschedparam()`, `posix_spawnattr_setsigmask()`,

29067 *posix_spawnp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sched.h>, <spawn.h>

29068 CHANGE HISTORY

29069 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29070 **NAME**

29071 `posix_spawnattr_getsigdefault`, `posix_spawnattr_setsigdefault` — get and set the spawn-
 29072 sigdefault attribute of a spawn attributes object (**ADVANCED REALTIME**)

29073 **SYNOPSIS**

29074 SPN `#include <signal.h>`

29075 `#include <spawn.h>`

```
29076 int posix_spawnattr_getsigdefault(const posix_spawnattr_t *
29077     restrict attr, sigset_t *restrict sigdefault);
29078 int posix_spawnattr_setsigdefault(posix_spawnattr_t *restrict attr,
29079     const sigset_t *restrict sigdefault);
29080
```

29081 **DESCRIPTION**

29082 The `posix_spawnattr_getsigdefault()` function shall obtain the value of the *spawn-sigdefault*
 29083 attribute from the attributes object referenced by *attr*.

29084 The `posix_spawnattr_setsigdefault()` function shall set the *spawn-sigdefault* attribute in an
 29085 initialized attributes object referenced by *attr*.

29086 The *spawn-sigdefault* attribute represents the set of signals to be forced to default signal handling
 29087 in the new process image (if `POSIX_SPAWN_SETSIGDEF` is set in the *spawn-flags* attribute) by a
 29088 spawn operation. The default value of this attribute shall be an empty signal set.

29089 **RETURN VALUE**

29090 Upon successful completion, `posix_spawnattr_getsigdefault()` shall return zero and store the value
 29091 of the *spawn-sigdefault* attribute of *attr* into the object referenced by the *sigdefault* parameter;
 29092 otherwise, an error number shall be returned to indicate the error.

29093 Upon successful completion, `posix_spawnattr_setsigdefault()` shall return zero; otherwise, an error
 29094 number shall be returned to indicate the error.

29095 **ERRORS**

29096 These functions may fail if:

29097 [EINVAL] The value specified by *attr* is invalid.

29098 The `posix_spawnattr_setsigdefault()` function may fail if:

29099 [EINVAL] The value of the attribute being set is not valid.

29100 **EXAMPLES**

29101 None.

29102 **APPLICATION USAGE**

29103 These functions are part of the Spawn option and need not be provided on all implementations.

29104 **RATIONALE**

29105 None.

29106 **FUTURE DIRECTIONS**

29107 None.

29108 **SEE ALSO**

29109 `posix_spawn()`, `posix_spawnattr_destroy()`, `posix_spawnattr_init()`, `posix_spawnattr_getflags()`,
 29110 `posix_spawnattr_getpgroup()`, `posix_spawnattr_getschedparam()`, `posix_spawnattr_getschedpolicy()`,
 29111 `posix_spawnattr_getsigmask()`, `posix_spawnattr_setflags()`, `posix_spawnattr_setpgroup()`,
 29112 `posix_spawnattr_setschedparam()`, `posix_spawnattr_setschedpolicy()`, `posix_spawnattr_setsigmask()`,
 29113 `posix_spawnnp()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<signal.h>`, `<spawn.h>`

29114 CHANGE HISTORY

29115 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29116 **NAME**

29117 `posix_spawnattr_getsigmask`, `posix_spawnattr_setsigmask` — get and set the spawn-sigmask
 29118 attribute of a spawn attributes object (**ADVANCED REALTIME**)

29119 **SYNOPSIS**

```
29120 SPN    #include <signal.h>
29121        #include <spawn.h>

29122        int posix_spawnattr_getsigmask(const posix_spawnattr_t *restrict attr,
29123                                       sigset_t *restrict sigmask);
29124        int posix_spawnattr_setsigmask(posix_spawnattr_t *restrict attr,
29125                                       const sigset_t *restrict sigmask);
29126
```

29127 **DESCRIPTION**

29128 The `posix_spawnattr_getsigmask()` function shall obtain the value of the *spawn-sigmask* attribute
 29129 from the attributes object referenced by *attr*.

29130 The `posix_spawnattr_setsigmask()` function shall set the *spawn-sigmask* attribute in an initialized
 29131 attributes object referenced by *attr*.

29132 The *spawn-sigmask* attribute represents the signal mask in effect in the new process image of a
 29133 spawn operation (if `POSIX_SPAWN_SETSIGMASK` is set in the *spawn-flags* attribute). The
 29134 default value of this attribute is unspecified.

29135 **RETURN VALUE**

29136 Upon successful completion, `posix_spawnattr_getsigmask()` shall return zero and store the value
 29137 of the *spawn-sigmask* attribute of *attr* into the object referenced by the *sigmask* parameter;
 29138 otherwise, an error number shall be returned to indicate the error.

29139 Upon successful completion, `posix_spawnattr_setsigmask()` shall return zero; otherwise, an error
 29140 number shall be returned to indicate the error.

29141 **ERRORS**

29142 These functions may fail if:

29143 [EINVAL] The value specified by *attr* is invalid.

29144 The `posix_spawnattr_setsigmask()` function may fail if:

29145 [EINVAL] The value of the attribute being set is not valid.

29146 **EXAMPLES**

29147 None.

29148 **APPLICATION USAGE**

29149 These functions are part of the Spawn option and need not be provided on all implementations.

29150 **RATIONALE**

29151 None.

29152 **FUTURE DIRECTIONS**

29153 None.

29154 **SEE ALSO**

29155 `posix_spawn()`, `posix_spawnattr_destroy()`, `posix_spawnattr_init()`, `posix_spawnattr_getsigdefault()`,
 29156 `posix_spawnattr_getflags()`, `posix_spawnattr_getpgroup()`, `posix_spawnattr_getschedparam()`,
 29157 `posix_spawnattr_getschedpolicy()`, `posix_spawnattr_setsigdefault()`, `posix_spawnattr_setflags()`,
 29158 `posix_spawnattr_setpgroup()`, `posix_spawnattr_setschedparam()`, `posix_spawnattr_setschedpolicy()`,
 29159 `posix_spawnnp()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<signal.h>`, `<spawn.h>`

29160 CHANGE HISTORY

29161 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29162 NAME

29163 `posix_spawnattr_init` — initialize the spawn attributes object (**ADVANCED REALTIME**)

29164 SYNOPSIS

29165 SPN `#include <spawn.h>`

29166 `int posix_spawnattr_init(posix_spawnattr_t *attr);`

29167

29168 DESCRIPTION

29169 Refer to *posix_spawnattr_destroy()*.

29170 NAME

29171 `posix_spawnattr_setflags` — set the spawn-flags attribute of a spawn attributes object
29172 (ADVANCED REALTIME)

29173 SYNOPSIS

29174 SPN `#include <spawn.h>`

29175 `int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags);`

29176

29177 DESCRIPTION

29178 Refer to *posix_spawnattr_getflags()*.

29179 **NAME**

29180 `posix_spawnattr_setpgroup` — set the spawn-pgroup attribute of a spawn attributes object
29181 (**ADVANCED REALTIME**)

29182 **SYNOPSIS**

29183 SPN `#include <spawn.h>`

29184 `int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup);`

29185

29186 **DESCRIPTION**

29187 Refer to *posix_spawnattr_getpgroup()*.

29188 NAME

29189 posix_spawnattr_setschedparam — set the spawn-schedparam attribute of a spawn attributes
29190 object (**ADVANCED REALTIME**)

29191 SYNOPSIS

29192 SPN PS #include <sched.h>

29193 #include <spawn.h>

```
29194       int posix_spawnattr_setschedparam(posix_spawnattr_t *restrict attr,  
29195                                        const struct sched_param *restrict schedparam);
```

29196

29197 DESCRIPTION

29198 Refer to *posix_spawnattr_getschedparam()*.

29199 **NAME**

29200 posix_spawnattr_setschedpolicy — set the spawn-schedpolicy attribute of a spawn attributes
29201 object (**ADVANCED REALTIME**)

29202 **SYNOPSIS**

29203 SPN PS #include <sched.h>

29204 #include <spawn.h>

29205 int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr,
29206 int schedpolicy);

29207

29208 **DESCRIPTION**

29209 Refer to *posix_spawnattr_getschedpolicy()*.

29210 NAME

29211 `posix_spawnattr_setsigdefault` — set the spawn-sigdefault attribute of a spawn attributes object
29212 (ADVANCED REALTIME)

29213 SYNOPSIS

29214 SPN `#include <signal.h>`

29215 `#include <spawn.h>`

```
29216        int posix_spawnattr_setsigdefault(posix_spawnattr_t *restrict attr,  
29217            const sigset_t *restrict sigdefault);
```

29218

29219 DESCRIPTION

29220 Refer to `posix_spawnattr_getsigdefault()`.

29221 **NAME**

29222 posix_spawnattr_setsigmask — set the spawn-sigmask attribute of a spawn attributes object
29223 **(ADVANCED REALTIME)**

29224 **SYNOPSIS**

29225 SPN #include <signal.h>

29226 #include <spawn.h>

29227 int posix_spawnattr_setsigmask(posix_spawnattr_t *restrict attr,
29228 const sigset_t *restrict sigmask);

29229

29230 **DESCRIPTION**

29231 Refer to *posix_spawnattr_getsigmask()*.

29232 NAME

29233 `posix_spawnnp` — spawn a process (**ADVANCED REALTIME**)

29234 SYNOPSIS

29235 SPN `#include <spawn.h>`

```
29236        int posix_spawnnp(pid_t *restrict pid, const char *restrict file,  
29237            const posix_spawn_file_actions_t *file_actions,  
29238            const posix_spawnattr_t *restrict attrp,  
29239            char *const argv[restrict], char *const envp[restrict]);  
29240
```

29241 DESCRIPTION

29242 Refer to *posix_spawn()*.

29243 **NAME**

29244 `posix_trace_attr_destroy`, `posix_trace_attr_init` — destroy and initialize the trace stream
 29245 attributes object (**TRACING**)

29246 **SYNOPSIS**

29247 TRC `#include <trace.h>`

29248 `int posix_trace_attr_destroy(trace_attr_t *attr);`

29249 `int posix_trace_attr_init(trace_attr_t *attr);`

29250

29251 **DESCRIPTION**

29252 The `posix_trace_attr_destroy()` function shall destroy an initialized trace attributes object. A
 29253 destroyed `attr` attributes object can be reinitialized using `posix_trace_attr_init()`; the results of
 29254 otherwise referencing the object after it has been destroyed are undefined.

29255 The `posix_trace_attr_init()` function shall initialize a trace attributes object `attr` with the default
 29256 value for all of the individual attributes used by a given implementation. The read-only
 29257 *generation-version* and *clock-resolution* attributes of the newly initialized trace attributes object
 29258 shall be set to their appropriate values (see Section 2.11.1.2 (on page 75)).

29259 Results are undefined if `posix_trace_attr_init()` is called specifying an already initialized `attr`
 29260 attributes object.

29261 Implementations may add extensions to the trace attributes object structure as permitted in the
 29262 Base Definitions volume of IEEE Std 1003.1-2001, Chapter 2, Conformance.

29263 The resulting attributes object (possibly modified by setting individual attributes values), when
 29264 used by `posix_trace_create()`, defines the attributes of the trace stream created. A single attributes
 29265 object can be used in multiple calls to `posix_trace_create()`. After one or more trace streams have
 29266 been created using an attributes object, any function affecting that attributes object, including
 29267 destruction, shall not affect any trace stream previously created. An initialized attributes object
 29268 also serves to receive the attributes of an existing trace stream or trace log when calling the
 29269 `posix_trace_get_attr()` function.

29270 **RETURN VALUE**

29271 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
 29272 return the corresponding error number.

29273 **ERRORS**

29274 The `posix_trace_attr_destroy()` function may fail if:

29275 [EINVAL] The value of `attr` is invalid.

29276 The `posix_trace_attr_init()` function shall fail if:

29277 [ENOMEM] Insufficient memory exists to initialize the trace attributes object.

29278 **EXAMPLES**

29279 None.

29280 **APPLICATION USAGE**

29281 None.

29282 **RATIONALE**

29283 None.

29284 FUTURE DIRECTIONS

29285 None.

29286 SEE ALSO

29287 *posix_trace_create()*, *posix_trace_get_attr()*, *uname()*, the Base Definitions volume of
29288 IEEE Std 1003.1-2001, <trace.h>

29289 CHANGE HISTORY

29290 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

29291 IEEE PASC Interpretation 1003.1 #123 is applied.

29292 **NAME**

29293 posix_trace_attr_getclockres, posix_trace_attr_getcreatetime, posix_trace_attr_getgenversion,
 29294 posix_trace_attr_getname, posix_trace_attr_setname — retrieve and set information about a
 29295 trace stream (**TRACING**)

29296 **SYNOPSIS**

```

29297 TRC    #include <time.h>
29298        #include <trace.h>

29299        int posix_trace_attr_getclockres(const trace_attr_t *attr,
29300            struct timespec *resolution);
29301        int posix_trace_attr_getcreatetime(const trace_attr_t *attr,
29302            struct timespec *createtime);

29303        #include <trace.h>

29304        int posix_trace_attr_getgenversion(const trace_attr_t *attr,
29305            char *genversion);
29306        int posix_trace_attr_getname(const trace_attr_t *attr,
29307            char *tracename);
29308        int posix_trace_attr_setname(trace_attr_t *attr,
29309            const char *tracename);
29310
```

29311 **DESCRIPTION**

29312 The *posix_trace_attr_getclockres()* function shall copy the clock resolution of the clock used to
 29313 generate timestamps from the *clock-resolution* attribute of the attributes object pointed to by the
 29314 *attr* argument into the structure pointed to by the *resolution* argument.

29315 The *posix_trace_attr_getcreatetime()* function shall copy the trace stream creation time from the
 29316 *creation-time* attribute of the attributes object pointed to by the *attr* argument into the structure
 29317 pointed to by the *createtime* argument. The *creation-time* attribute shall represent the time of
 29318 creation of the trace stream.

29319 The *posix_trace_attr_getgenversion()* function shall copy the string containing version information
 29320 from the *generation-version* attribute of the attributes object pointed to by the *attr* argument into
 29321 the string pointed to by the *genversion* argument. The *genversion* argument shall be the address of
 29322 a character array which can store at least {TRACE_NAME_MAX} characters.

29323 The *posix_trace_attr_getname()* function shall copy the string containing the trace name from the
 29324 *trace-name* attribute of the attributes object pointed to by the *attr* argument into the string
 29325 pointed to by the *tracename* argument. The *tracename* argument shall be the address of a character
 29326 array which can store at least {TRACE_NAME_MAX} characters.

29327 The *posix_trace_attr_setname()* function shall set the name in the *trace-name* attribute of the
 29328 attributes object pointed to by the *attr* argument, using the trace name string supplied by the
 29329 *tracename* argument. If the supplied string contains more than {TRACE_NAME_MAX}
 29330 characters, the name copied into the *trace-name* attribute may be truncated to one less than the
 29331 length of {TRACE_NAME_MAX} characters. The default value is a null string.

29332 **RETURN VALUE**

29333 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
 29334 return the corresponding error number.

29335 If successful, the *posix_trace_attr_getclockres()* function stores the *clock-resolution* attribute value
 29336 in the object pointed to by *resolution*. Otherwise, the content of this object is unspecified.

29337 If successful, the *posix_trace_attr_getcreatetime()* function stores the trace stream creation time in
 29338 the object pointed to by *createtime*. Otherwise, the content of this object is unspecified.

29339 If successful, the *posix_trace_attr_getgenversion()* function stores the trace version information in
 29340 the string pointed to by *genversion*. Otherwise, the content of this string is unspecified.

29341 If successful, the *posix_trace_attr_getname()* function stores the trace name in the string pointed
 29342 to by *tracename*. Otherwise, the content of this string is unspecified.

29343 ERRORS

29344 The *posix_trace_attr_getclockres()*, *posix_trace_attr_getcreatetime()*, *posix_trace_attr_getgenversion()*,
 29345 and *posix_trace_attr_getname()* functions may fail if:

29346 [EINVAL] The value specified by one of the arguments is invalid.

29347 EXAMPLES

29348 None.

29349 APPLICATION USAGE

29350 None.

29351 RATIONALE

29352 None.

29353 FUTURE DIRECTIONS

29354 None.

29355 SEE ALSO

29356 *posix_trace_attr_init()*, *posix_trace_create()*, *posix_trace_get_attr()*, *uname()*, the Base Definitions
 29357 volume of IEEE Std 1003.1-2001, <time.h>, <trace.h>

29358 CHANGE HISTORY

29359 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

29360 NAME

29361 posix_trace_attr_getinherited, posix_trace_attr_getlogfullpolicy,
 29362 posix_trace_attr_getstreamfullpolicy, posix_trace_attr_setinherited,
 29363 posix_trace_attr_setlogfullpolicy, posix_trace_attr_setstreamfullpolicy — retrieve and set the
 29364 behavior of a trace stream (**TRACING**)

29365 SYNOPSIS

```
29366 TRC      #include <trace.h>

29367 TRC TRI   int posix_trace_attr_getinherited(const trace_attr_t *restrict attr,
29368                                     int *restrict inheritancepolicy);
29369 TRC TRL   int posix_trace_attr_getlogfullpolicy(const trace_attr_t *restrict attr,
29370                                     int *restrict logpolicy);
29371 TRC       int posix_trace_attr_getstreamfullpolicy(const trace_attr_t *attr,
29372                                     int *streampolicy);
29373 TRC TRI   int posix_trace_attr_setinherited(trace_attr_t *attr,
29374                                     int inheritancepolicy);
29375 TRC TRL   int posix_trace_attr_setlogfullpolicy(trace_attr_t *attr,
29376                                     int logpolicy);
29377 TRC       int posix_trace_attr_setstreamfullpolicy(trace_attr_t *attr,
29378                                     int streampolicy);
29379
```

29380 DESCRIPTION

29381 TRI The *posix_trace_attr_getinherited()* and *posix_trace_attr_setinherited()* functions, respectively, shall
 29382 get and set the inheritance policy stored in the *inheritance* attribute for traced processes across
 29383 the *fork()* and *spawn()* operations. The *inheritance* attribute of the attributes object pointed to by
 29384 the *attr* argument shall be set to one of the following values defined by manifest constants in the
 29385 **<trace.h>** header:

29386 POSIX_TRACE_CLOSE_FOR_CHILD

29387 After a *fork()* or *spawn()* operation, the child shall not be traced, and tracing of the parent
 29388 shall continue.

29389 POSIX_TRACE_INHERITED

29390 After a *fork()* or *spawn()* operation, if the parent is being traced, its child shall be
 29391 concurrently traced using the same trace stream.

29392 The default value for the *inheritance* attribute is **POSIX_TRACE_CLOSE_FOR_CHILD**.

29393 TRL The *posix_trace_attr_getlogfullpolicy()* and *posix_trace_attr_setlogfullpolicy()* functions,
 29394 respectively, shall get and set the trace log full policy stored in the *log-full-policy* attribute of the
 29395 attributes object pointed to by the *attr* argument.

29396 The *log-full-policy* attribute shall be set to one of the following values defined by manifest
 29397 constants in the **<trace.h>** header:

29398 POSIX_TRACE_LOOP

29399 The trace log shall loop until the associated trace stream is stopped. This policy means that
 29400 when the trace log gets full, the file system shall reuse the resources allocated to the oldest
 29401 trace events that were recorded. In this way, the trace log will always contain the most
 29402 recent trace events flushed.

29403 POSIX_TRACE_UNTIL_FULL

29404 The trace stream shall be flushed to the trace log until the trace log is full. This condition can
 29405 be deduced from the *posix_log_full_status* member status (see the **posix_trace_status_info**
 29406 structure defined in **<trace.h>**). The last recorded trace event shall be the

29407	POSIX_TRACE_STOP trace event.
29408	POSIX_TRACE_APPEND
29409	The associated trace stream shall be flushed to the trace log without log size limitation. If
29410	the application specifies POSIX_TRACE_APPEND, the implementation shall ignore the
29411	<i>log-max-size</i> attribute.
29412	The default value for the <i>log-full-policy</i> attribute is POSIX_TRACE_LOOP.
29413	The <i>posix_trace_attr_getstreamfullpolicy()</i> and <i>posix_trace_attr_setstreamfullpolicy()</i> functions,
29414	respectively, shall get and set the trace stream full policy stored in the <i>stream-full-policy</i> attribute
29415	of the attributes object pointed to by the <i>attr</i> argument.
29416	The <i>stream-full-policy</i> attribute shall be set to one of the following values defined by manifest
29417	constants in the <trace.h> header:
29418	POSIX_TRACE_LOOP
29419	The trace stream shall loop until explicitly stopped by the <i>posix_trace_stop()</i> function. This
29420	policy means that when the trace stream is full, the trace system shall reuse the resources
29421	allocated to the oldest trace events recorded. In this way, the trace stream will always
29422	contain the most recent trace events recorded.
29423	POSIX_TRACE_UNTIL_FULL
29424	The trace stream will run until the trace stream resources are exhausted. Then the trace
29425	stream will stop. This condition can be deduced from <i>posix_stream_status</i> and
29426	<i>posix_stream_full_status</i> (see the posix_trace_status_info structure defined in <trace.h>).
29427	When this trace stream is read, a POSIX_TRACE_STOP trace event shall be reported after
29428	reporting the last recorded trace event. The trace system shall reuse the resources allocated
29429	to any trace events already reported—see the <i>posix_trace_getnext_event()</i> ,
29430	<i>posix_trace_trygetnext_event()</i> , and <i>posix_trace_timedgetnext_event()</i> functions—or already
29431	flushed for an active trace stream with log if the Trace Log option is supported; see the
29432	<i>posix_trace_flush()</i> function. The trace system shall restart the trace stream when it is empty
29433	and may restart it sooner. A POSIX_TRACE_START trace event shall be reported before
29434	reporting the next recorded trace event.
29435 TRL	POSIX_TRACE_FLUSH
29436	If the Trace Log option is supported, this policy is identical to the
29437	POSIX_TRACE_UNTIL_FULL trace stream full policy except that the trace stream shall be
29438	flushed regularly as if <i>posix_trace_flush()</i> had been explicitly called. Defining this policy for
29439	an active trace stream without log shall be invalid.
29440	The default value for the <i>stream-full-policy</i> attribute shall be POSIX_TRACE_LOOP for an active
29441	trace stream without log.
29442 TRL	If the Trace Log option is supported, the default value for the <i>stream-full-policy</i> attribute shall be
29443	POSIX_TRACE_FLUSH for an active trace stream with log.
29444	RETURN VALUE
29445	Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
29446	return the corresponding error number.
29447 TRI	If successful, the <i>posix_trace_attr_getinherited()</i> function shall store the <i>inheritance</i> attribute value
29448	in the object pointed to by <i>inheritancepolicy</i> . Otherwise, the content of this object is undefined.
29449 TRL	If successful, the <i>posix_trace_attr_getlogfullpolicy()</i> function shall store the <i>log-full-policy</i> attribute
29450	value in the object pointed to by <i>logpolicy</i> . Otherwise, the content of this object is undefined.
29451	If successful, the <i>posix_trace_attr_getstreamfullpolicy()</i> function shall store the <i>stream-full-policy</i>
29452	attribute value in the object pointed to by <i>streampolicy</i> . Otherwise, the content of this object is

29453 undefined.

29454 **ERRORS**

29455 These functions may fail if:

29456 [EINVAL] The value specified by at least one of the arguments is invalid.

29457 **EXAMPLES**

29458 None.

29459 **APPLICATION USAGE**

29460 None.

29461 **RATIONALE**

29462 None.

29463 **FUTURE DIRECTIONS**

29464 None.

29465 **SEE ALSO**

29466 *fork()*, *posix_trace_attr_init()*, *posix_trace_create()*, *posix_trace_flush()*, *posix_trace_get_attr()*,
29467 *posix_trace_getnext_event()*, *posix_trace_start()*, *posix_trace_timedgetnext_event()*, the Base
29468 Definitions volume of IEEE Std 1003.1-2001, <trace.h>

29469 **CHANGE HISTORY**

29470 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

29471 NAME

29472 posix_trace_attr_getlogsize, posix_trace_attr_getmaxdatasize,
29473 posix_trace_attr_getmaxsystemeventsizesize, posix_trace_attr_getmaxusereventsizesize,
29474 posix_trace_attr_getstreamsize, posix_trace_attr_setlogsize, posix_trace_attr_setmaxdatasize,
29475 posix_trace_attr_setstreamsize — retrieve and set trace stream size attributes (TRACING)

29476 SYNOPSIS

```
29477 TRC    #include <sys/types.h>
29478          #include <trace.h>

29479 TRC TRL int posix_trace_attr_getlogsize(const trace_attr_t *restrict attr,
29480          size_t *restrict logsize);
29481 TRC    int posix_trace_attr_getmaxdatasize(const trace_attr_t *restrict attr,
29482          size_t *restrict maxdatasize);
29483          int posix_trace_attr_getmaxsystemeventsizesize(
29484          const trace_attr_t *restrict attr,
29485          size_t *restrict eventsizesize);
29486          int posix_trace_attr_getmaxusereventsizesize(
29487          const trace_attr_t *restrict attr,
29488          size_t data_len, size_t *restrict eventsizesize);
29489          int posix_trace_attr_getstreamsize(const trace_attr_t *restrict attr,
29490          size_t *restrict streamsize);
29491 TRC TRL int posix_trace_attr_setlogsize(trace_attr_t *attr,
29492          size_t logsize);
29493 TRC    int posix_trace_attr_setmaxdatasize(trace_attr_t *attr,
29494          size_t maxdatasize);
29495          int posix_trace_attr_setstreamsize(trace_attr_t *attr,
29496          size_t streamsize);
29497
```

29498 DESCRIPTION

29499 TRL The *posix_trace_attr_getlogsize()* function shall copy the log size, in bytes, from the *log-max-size*
29500 attribute of the attributes object pointed to by the *attr* argument into the variable pointed to by
29501 the *logsize* argument. This log size is the maximum total of bytes that shall be allocated for
29502 system and user trace events in the trace log. The default value for the *log-max-size* attribute is
29503 implementation-defined.

29504 The *posix_trace_attr_setlogsize()* function shall set the maximum allowed size, in bytes, in the
29505 *log-max-size* attribute of the attributes object pointed to by the *attr* argument, using the size value
29506 supplied by the *logsize* argument.

29507 The trace log size shall be used if the *log-full-policy* attribute is set to POSIX_TRACE_LOOP or
29508 POSIX_TRACE_UNTIL_FULL. If the *log-full-policy* attribute is set to POSIX_TRACE_APPEND,
29509 the implementation shall ignore the *log-max-size* attribute.

29510 The *posix_trace_attr_getmaxdatasize()* function shall copy the maximum user trace event data
29511 size, in bytes, from the *max-data-size* attribute of the attributes object pointed to by the *attr*
29512 argument into the variable pointed to by the *maxdatasize* argument. The default value for the
29513 *max-data-size* attribute is implementation-defined.

29514 The *posix_trace_attr_getmaxsystemeventsizesize()* function shall calculate the maximum memory size,
29515 in bytes, required to store a single system trace event. This value is calculated for the trace
29516 stream attributes object pointed to by the *attr* argument and is returned in the variable pointed
29517 to by the *eventsizesize* argument.

29518 The values returned as the maximum memory sizes of the user and system trace events shall be
 29519 such that if the sum of the maximum memory sizes of a set of the trace events that may be
 29520 recorded in a trace stream is less than or equal to the *stream-min-size* attribute of that trace
 29521 stream, the system provides the necessary resources for recording all those trace events, without
 29522 loss.

29523 The *posix_trace_attr_getmaxusereventsize()* function shall calculate the maximum memory size, in
 29524 bytes, required to store a single user trace event generated by a call to *posix_trace_event()* with a
 29525 *data_len* parameter equal to the *data_len* value specified in this call. This value is calculated for
 29526 the trace stream attributes object pointed to by the *attr* argument and is returned in the variable
 29527 pointed to by the *eventsize* argument.

29528 The *posix_trace_attr_getstreamsize()* function shall copy the stream size, in bytes, from the
 29529 *stream-min-size* attribute of the attributes object pointed to by the *attr* argument into the variable
 29530 pointed to by the *streamsize* argument.

29531 This stream size is the current total memory size reserved for system and user trace events in the
 29532 trace stream. The default value for the *stream-min-size* attribute is implementation-defined. The
 29533 stream size refers to memory used to store trace event records. Other stream data (for example,
 29534 trace attribute values) shall not be included in this size.

29535 The *posix_trace_attr_setmaxdatasize()* function shall set the maximum allowed size, in bytes, in
 29536 the *max-data-size* attribute of the attributes object pointed to by the *attr* argument, using the size
 29537 value supplied by the *maxdatasize* argument. This maximum size is the maximum allowed size
 29538 for the user data argument which may be passed to *posix_trace_event()*. The implementation
 29539 shall be allowed to truncate data passed to *trace_user_event* which is longer than *maxdatasize*.

29540 The *posix_trace_attr_setstreamsize()* function shall set the minimum allowed size, in bytes, in the
 29541 *stream-min-size* attribute of the attributes object pointed to by the *attr* argument, using the size
 29542 value supplied by the *streamsize* argument.

29543 RETURN VALUE

29544 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
 29545 return the corresponding error number.

29546 TRL The *posix_trace_attr_getlogsize()* function stores the maximum trace log allowed size in the object
 29547 pointed to by *logsize*, if successful.

29548 The *posix_trace_attr_getmaxdatasize()* function stores the maximum trace event record memory
 29549 size in the object pointed to by *maxdatasize*, if successful.

29550 The *posix_trace_attr_getmaxsystemeventsize()* function stores the maximum memory size to store
 29551 a single system trace event in the object pointed to by *eventsize*, if successful.

29552 The *posix_trace_attr_getmaxusereventsize()* function stores the maximum memory size to store a
 29553 single user trace event in the object pointed to by *eventsize*, if successful.

29554 The *posix_trace_attr_getstreamsize()* function stores the maximum trace stream allowed size in
 29555 the object pointed to by *streamsize*, if successful.

29556 ERRORS

29557 These functions may fail if:

29558 [EINVAL] The value specified by one of the arguments is invalid.

29559 EXAMPLES

29560 None.

29561 APPLICATION USAGE

29562 None.

29563 RATIONALE

29564 None.

29565 FUTURE DIRECTIONS

29566 None.

29567 SEE ALSO

29568 *posix_trace_attr_init()*, *posix_trace_create()*, *posix_trace_event()*, *posix_trace_get_attr()*, the Base

29569 Definitions volume of IEEE Std 1003.1-2001, <**sys/types.h**>, <**trace.h**>

29570 CHANGE HISTORY

29571 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

29572 NAME

29573 `posix_trace_attr_getname` — retrieve and set information about a trace stream (**TRACING**)

29574 SYNOPSIS

29575 TRC `#include <trace.h>`

29576 `int posix_trace_attr_getname(const trace_attr_t *attr,`
29577 `char *tracename);`

29578

29579 DESCRIPTION

29580 Refer to `posix_trace_attr_getclockres()`.

29581 NAME

29582 posix_trace_attr_getstreamfullpolicy — retrieve and set the behavior of a trace stream
29583 (TRACING)

29584 SYNOPSIS

29585 TRC #include <trace.h>

```
29586       int posix_trace_attr_getstreamfullpolicy(const trace_attr_t *attr,  
29587                                               int *streampolicy);
```

29588

29589 DESCRIPTION

29590 Refer to *posix_trace_attr_getinherited()*.

29591 **NAME**29592 **posix_trace_attr_getstreamsize** — retrieve and set trace stream size attributes (**TRACING**)29593 **SYNOPSIS**29594 TRC `#include <sys/types.h>`29595 `#include <trace.h>`29596 `int posix_trace_attr_getstreamsize(const trace_attr_t *restrict attr,`
29597 `size_t *restrict streamsize);`

29598

29599 **DESCRIPTION**29600 Refer to *posix_trace_attr_getlogsize()*.

29601 NAME

29602 posix_trace_attr_init — initialize the trace stream attributes object (**TRACING**)

29603 SYNOPSIS

29604 TRC #include <trace.h>

29605 int posix_trace_attr_init(trace_attr_t *attr);

29606

29607 DESCRIPTION

29608 Refer to *posix_trace_attr_destroy()*.

29609 **NAME**

29610 posix_trace_attr_setinherited, posix_trace_attr_setlogfullpolicy — retrieve and set the behavior
29611 of a trace stream (**TRACING**)

29612 **SYNOPSIS**

29613 TRC #include <trace.h>

```
29614 TRC TRI  int posix_trace_attr_setinherited(trace_attr_t *attr,  
29615           int inheritancepolicy);  
29616           int posix_trace_attr_setlogfullpolicy(trace_attr_t *attr,  
29617           int logpolicy);  
29618
```

29619 **DESCRIPTION**

29620 Refer to *posix_trace_attr_getinherited()*.

29622 `posix_trace_attr_setlogsize`, `posix_trace_attr_setmaxdatasize` — retrieve and set trace stream
29623 size attributes (**TRACING**)

29624 **SYNOPSIS**

```
29625 TRC      #include <sys/types.h>
```

```
29626 #include <trace.h>
```

```
29627 TRC   TRL   int posix_trace_attr_setlogsize(trace_attr_t *attr,
29628                size_t logsize);
```

```
29629 TRC      int posix_trace_attr_setmaxdatasize(trace_attr_t *attr,
29630              size_t maxdatasize);
```

29632 DESCRIPTION

29633 Refer to *posix_trace_attr_getlogsize()*.

29634 **NAME**

29635 posix_trace_attr_setname — retrieve and set information about a trace stream (**TRACING**)

29636 **SYNOPSIS**

29637 TRC #include <trace.h>

29638 int posix_trace_attr_setname(trace_attr_t *attr,
29639 const char *tracename);

29640

29641 **DESCRIPTION**

29642 Refer to *posix_trace_attr_getclockres()*.

29643 NAME

29644 posix_trace_attr_setstreamfullpolicy — retrieve and set the behavior of a trace stream
29645 (TRACING)

29646 SYNOPSIS

29647 TRC #include <trace.h>

29648 TRC TRL int posix_trace_attr_setlogfullpolicy(trace_attr_t *attr,
29649 int logpolicy);

29650

29651 DESCRIPTION

29652 Refer to *posix_trace_attr_getinherited()*.

29653 **NAME**29654 posix_trace_attr_setstreamsize — retrieve and set trace stream size attributes (**TRACING**)29655 **SYNOPSIS**

29656 TRC #include <sys/types.h>

29657 #include <trace.h>

29658 int posix_trace_attr_setstreamsize(trace_attr_t *attr,
29659 size_t streamsize);

29660

29661 **DESCRIPTION**29662 Refer to *posix_trace_attr_getlogsize()*.

29663 NAME

29664 `posix_trace_clear` — clear trace stream and trace log (**TRACING**)

29665 SYNOPSIS

```
29666 TRC        #include <sys/types.h>
29667               #include <trace.h>
```

```
29668               int posix_trace_clear(trace_id_t trid);
29669
```

29670 DESCRIPTION

29671 The *posix_trace_clear()* function shall reinitialize the trace stream identified by the argument *trid* as if it were returning from the *posix_trace_create()* function, except that the same allocated resources shall be reused, the mapping of trace event type identifiers to trace event names shall be unchanged, and the trace stream status shall remain unchanged (that is, if it was running, it remains running and if it was suspended, it remains suspended).

29676 All trace events in the trace stream recorded before the call to *posix_trace_clear()* shall be lost. The *posix_stream_full_status* status shall be set to `POSIX_TRACE_NOT_FULL`. There is no guarantee that all trace events that occurred during the *posix_trace_clear()* call are recorded; the behavior with respect to trace points that may occur during this call is unspecified.

29680 TRL If the Trace Log option is supported and the trace stream has been created with a log, the *posix_trace_clear()* function shall reinitialize the trace stream with the same behavior as if the trace stream was created without the log, plus it shall reinitialize the trace log associated with the trace stream identified by the argument *trid* as if it were returning from the *posix_trace_create_withlog()* function, except that the same allocated resources, for the trace log, may be reused and the associated trace stream status remains unchanged. The first trace event recorded in the trace log after the call to *posix_trace_clear()* shall be the same as the first trace event recorded in the active trace stream after the call to *posix_trace_clear()*. The *posix_log_full_status* status shall be set to `POSIX_TRACE_NOT_FULL`. There is no guarantee that all trace events that occurred during the *posix_trace_clear()* call are recorded in the trace log; the behavior with respect to trace points that may occur during this call is unspecified. If the log full policy is `POSIX_TRACE_APPEND`, the effect of a call to this function is unspecified for the trace log associated with the trace stream identified by the *trid* argument.

29693 RETURN VALUE

29694 Upon successful completion, the *posix_trace_clear()* function shall return a value of zero. Otherwise, it shall return the corresponding error number.

29696 ERRORS

29697 The *posix_trace_clear()* function shall fail if:

29698 [EINVAL] The value of the *trid* argument does not correspond to an active trace stream.

29699 EXAMPLES

29700 None.

29701 APPLICATION USAGE

29702 None.

29703 RATIONALE

29704 None.

29705 FUTURE DIRECTIONS

29706 None.

29707 **SEE ALSO**

29708 *posix_trace_attr_init()*, *posix_trace_create()*, *posix_trace_flush()*, *posix_trace_get_attr()*, the Base
29709 Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <trace.h>

29710 **CHANGE HISTORY**

29711 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

29712 IEEE PASC Interpretation 1003.1 #123 is applied.

29713 NAME

29714 `posix_trace_close`, `posix_trace_open`, `posix_trace_rewind` — trace log management (**TRACING**)

29715 SYNOPSIS

29716 TRC TRL `#include <trace.h>`

```
29717     int posix_trace_close(trace_id_t trid);
29718     int posix_trace_open(int file_desc, trace_id_t *trid);
29719     int posix_trace_rewind(trace_id_t trid);
29720
```

29721 DESCRIPTION

29722 The `posix_trace_close()` function shall deallocate the trace log identifier indicated by *trid*, and all
29723 of its associated resources. If there is no valid trace log pointed to by the *trid*, this function shall
29724 fail.

29725 The `posix_trace_open()` function shall allocate the necessary resources and establish the
29726 connection between a trace log identified by the *file_desc* argument and a trace stream identifier
29727 identified by the object pointed to by the *trid* argument. The *file_desc* argument should be a valid
29728 open file descriptor that corresponds to a trace log. The *file_desc* argument shall be open for
29729 reading. The current trace event timestamp, which specifies the timestamp of the trace event
29730 that will be read by the next call to `posix_trace_getnext_event()`, shall be set to the timestamp of
29731 the oldest trace event recorded in the trace log identified by *trid*.

29732 The `posix_trace_open()` function shall return a trace stream identifier in the variable pointed to by
29733 the *trid* argument, that may only be used by the following functions:

29734	<code>posix_trace_close()</code>	<code>posix_trace_get_attr()</code>
29735	<code>posix_trace_eventid_equal()</code>	<code>posix_trace_get_status()</code>
29736	<code>posix_trace_eventid_get_name()</code>	<code>posix_trace_getnext_event()</code>
29737	<code>posix_trace_eventtypelist_getnext_id()</code>	<code>posix_trace_rewind()</code>
29738	<code>posix_trace_eventtypelist_rewind()</code>	

29739 In particular, notice that the operations normally used by a trace controller process, such as
29740 `posix_trace_start()`, `posix_trace_stop()`, or `posix_trace_shutdown()`, cannot be invoked using the
29741 trace stream identifier returned by the `posix_trace_open()` function.

29742 The `posix_trace_rewind()` function shall reset the current trace event timestamp, which specifies
29743 the timestamp of the trace event that will be read by the next call to `posix_trace_getnext_event()`,
29744 to the timestamp of the oldest trace event recorded in the trace log identified by *trid*.

29745 RETURN VALUE

29746 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
29747 return the corresponding error number.

29748 If successful, the `posix_trace_open()` function stores the trace stream identifier value in the object
29749 pointed to by *trid*.

29750 ERRORS

29751 The `posix_trace_open()` function shall fail if:

29752	[EINTR]	The operation was interrupted by a signal and thus no trace log was opened.
29753	[EINVAL]	The object pointed to by <i>file_desc</i> does not correspond to a valid trace log.

29754 The `posix_trace_close()` and `posix_trace_rewind()` functions may fail if:

29755	[EINVAL]	The object pointed to by <i>trid</i> does not correspond to a valid trace log.
-------	----------	--

29756 **EXAMPLES**

29757 None.

29758 **APPLICATION USAGE**

29759 None.

29760 **RATIONALE**

29761 None.

29762 **FUTURE DIRECTIONS**

29763 None.

29764 **SEE ALSO**

29765 *posix_trace_get_attr()*, *posix_trace_get_filter()*, *posix_trace_getnext_event()*, the Base Definitions
29766 volume of IEEE Std 1003.1-2001, <**trace.h**>

29767 **CHANGE HISTORY**

29768 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

29769 IEEE PASC Interpretation 1003.1 #123 is applied.

29770 NAME

29771 `posix_trace_create`, `posix_trace_create_withlog`, `posix_trace_flush`, `posix_trace_shutdown` —
29772 trace stream initialization, flush, and shutdown from a process (TRACING)

29773 SYNOPSIS

```
29774 TRC    #include <sys/types.h>
29775        #include <trace.h>

29776        int posix_trace_create(pid_t pid,
29777                               const trace_attr_t *restrict attr,
29778                               trace_id_t *restrict trid);
29779 TRC TRL int posix_trace_create_withlog(pid_t pid,
29780                                       const trace_attr_t *restrict attr, int file_desc,
29781                                       trace_id_t *restrict trid);
29782        int posix_trace_flush(trace_id_t trid);
29783 TRC    int posix_trace_shutdown(trace_id_t trid);
29784
```

29785 DESCRIPTION

29786 The `posix_trace_create()` function shall create an active trace stream. It allocates all the resources
29787 needed by the trace stream being created for tracing the process specified by *pid* in accordance
29788 with the *attr* argument. The *attr* argument represents the initial attributes of the trace stream and
29789 shall have been initialized by the function `posix_trace_attr_init()` prior to the `posix_trace_create()`
29790 call. If the argument *attr* is NULL, the default attributes shall be used. The *attr* attributes object
29791 shall be manipulated through a set of functions described in the `posix_trace_attr` family of
29792 functions. If the attributes of the object pointed to by *attr* are modified later, the attributes of the
29793 trace stream shall not be affected. The *creation-time* attribute of the newly created trace stream
29794 shall be set to the value of the system clock, if the Timers option is not supported, or to the value
29795 of the CLOCK_REALTIME clock, if the Timers option is supported.

29796 The *pid* argument represents the target process to be traced. If the process executing this
29797 function does not have appropriate privileges to trace the process identified by *pid*, an error shall
29798 be returned. If the *pid* argument is zero, the calling process shall be traced.

29799 The `posix_trace_create()` function shall store the trace stream identifier of the new trace stream in
29800 the object pointed to by the *trid* argument. This trace stream identifier shall be used in
29801 subsequent calls to control tracing. The *trid* argument may only be used by the following
29802 functions:

29803	<code>posix_trace_clear()</code>	<code>posix_trace_getnext_event()</code>
29804	<code>posix_trace_eventid_equal()</code>	<code>posix_trace_shutdown()</code>
29805	<code>posix_trace_eventid_get_name()</code>	<code>posix_trace_start()</code>
29806	<code>posix_trace_eventtypelist_getnext_id()</code>	<code>posix_trace_stop()</code>
29807	<code>posix_trace_eventtypelist_rewind()</code>	<code>posix_trace_timedgetnext_event()</code>
29808	<code>posix_trace_get_attr()</code>	<code>posix_trace_trid_eventid_open()</code>
29809	<code>posix_trace_get_status()</code>	<code>posix_trace_trygetnext_event()</code>

29810 TEF If the Trace Event Filter option is supported, the following additional functions may use the *trid*
29811 argument:

```
29812 posix_trace_get_filter()    posix_trace_set_filter()
```

29813

29814 In particular, notice that the operations normally used by a trace analyzer process, such as
 29815 *posix_trace_rewind()* or *posix_trace_close()*, cannot be invoked using the trace stream identifier
 29816 returned by the *posix_trace_create()* function.

29817 TEF A trace stream shall be created in a suspended state. If the Trace Event Filter option is
 29818 supported, its trace event type filter shall be empty.

29819 The *posix_trace_create()* function may be called multiple times from the same or different
 29820 processes, with the system-wide limit indicated by the runtime invariant value
 29821 {TRACE_SYS_MAX}, which has the minimum value {_POSIX_TRACE_SYS_MAX}.

29822 The trace stream identifier returned by the *posix_trace_create()* function in the argument pointed
 29823 to by *trid* is valid only in the process that made the function call. If it is used from another
 29824 process, that is a child process, in functions defined in IEEE Std 1003.1-2001, these functions shall
 29825 return with the error [EINVAL].

29826 TRL The *posix_trace_create_withlog()* function shall be equivalent to *posix_trace_create()*, except that it
 29827 associates a trace log with this stream. The *file_desc* argument shall be the file descriptor
 29828 designating the trace log destination. The function shall fail if this file descriptor refers to a file
 29829 with a file type that is not compatible with the log policy associated with the trace log. The list of
 29830 the appropriate file types that are compatible with each log policy is implementation-defined.

29831 The *posix_trace_create_withlog()* function shall return in the parameter pointed to by *trid* the trace
 29832 stream identifier, which uniquely identifies the newly created trace stream, and shall be used in
 29833 subsequent calls to control tracing. The *trid* argument may only be used by the following
 29834 functions:

29835	<i>posix_trace_clear()</i>	<i>posix_trace_getnext_event()</i>
29836	<i>posix_trace_eventid_equal()</i>	<i>posix_trace_shutdown()</i>
29837	<i>posix_trace_eventid_get_name()</i>	<i>posix_trace_start()</i>
29838	<i>posix_trace_eventtypelist_getnext_id()</i>	<i>posix_trace_stop()</i>
29839	<i>posix_trace_eventtypelist_rewind()</i>	<i>posix_trace_timedgetnext_event()</i>
29840	<i>posix_trace_flush()</i>	<i>posix_trace_trid_eventid_open()</i>
29841	<i>posix_trace_get_attr()</i>	<i>posix_trace_trygetnext_event()</i>
29842	<i>posix_trace_get_status()</i>	

29843

29844 TRL TEF If the Trace Event Filter option is supported, the following additional functions may use the *trid*
 29845 argument:

29846 *posix_trace_get_filter()* *posix_trace_set_filter()*

29847

29848 TRL In particular, notice that the operations normally used by a trace analyzer process, such as
 29849 *posix_trace_rewind()* or *posix_trace_close()*, cannot be invoked using the trace stream identifier
 29850 returned by the *posix_trace_create_withlog()* function.

29851 The *posix_trace_flush()* function shall initiate a flush operation which copies the contents of the
 29852 trace stream identified by the argument *trid* into the trace log associated with the trace stream at
 29853 the creation time. If no trace log has been associated with the trace stream pointed to by *trid*, this
 29854 function shall return an error. The termination of the flush operation can be polled by the
 29855 *posix_trace_get_status()* function. During the flush operation, it shall be possible to trace new
 29856 trace events up to the point when the trace stream becomes full. After flushing is completed, the
 29857 space used by the flushed trace events shall be available for tracing new trace events.

29858	If flushing the trace stream causes the resulting trace log to become full, the trace log full policy shall be applied. If the trace <i>log-full-policy</i> attribute is set, the following occurs:	
29859		
29860	POSIX_TRACE_UNTIL_FULL	
29861	The trace events that have not yet been flushed shall be discarded.	
29862	POSIX_TRACE_LOOP	
29863	The trace events that have not yet been flushed shall be written to the beginning of the trace log, overwriting previous trace events stored there.	
29864		
29865	POSIX_TRACE_APPEND	
29866	The trace events that have not yet been flushed shall be appended to the trace log.	
29867		
29868	The <i>posix_trace_shutdown()</i> function shall stop the tracing of trace events in the trace stream identified by <i>trid</i> , as if <i>posix_trace_stop()</i> had been invoked. The <i>posix_trace_shutdown()</i> function shall free all the resources associated with the trace stream.	
29869		
29870		
29871	The <i>posix_trace_shutdown()</i> function shall not return until all the resources associated with the trace stream have been freed. When the <i>posix_trace_shutdown()</i> function returns, the <i>trid</i> argument becomes an invalid trace stream identifier. A call to this function shall unconditionally deallocate the resources regardless of whether all trace events have been retrieved by the analyzer process. Any thread blocked on one of the <i>trace_getnext_event()</i> functions (which specified this <i>trid</i>) before this call is unblocked with the error [EINVAL].	
29872		
29873		
29874		
29875		
29876		
29877	If the process exits, invokes a member of the <i>exec</i> family of functions, or is terminated, the trace streams that the process had created and that have not yet been shut down, shall be automatically shut down as if an explicit call were made to the <i>posix_trace_shutdown()</i> function.	
29878		
29879		
29880	TRL	For an active trace stream with log, when the <i>posix_trace_shutdown()</i> function is called, all trace events that have not yet been flushed to the trace log shall be flushed, as in the <i>posix_trace_flush()</i> function, and the trace log shall be closed.
29881		
29882		
29883	When a trace log is closed, all the information that may be retrieved later from the trace log through the trace interface shall have been written to the trace log. This information includes the trace attributes, the list of trace event types (with the mapping between trace event names and trace event type identifiers), and the trace status.	
29884		
29885		
29886		
29887	In addition, unspecified information shall be written to the trace log to allow detection of a valid trace log during the <i>posix_trace_open()</i> operation.	
29888		
29889	The <i>posix_trace_shutdown()</i> function shall not return until all trace events have been flushed.	
29890	RETURN VALUE	
29891	Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.	
29892		
29893	TRL	The <i>posix_trace_create()</i> and <i>posix_trace_create_withlog()</i> functions store the trace stream identifier value in the object pointed to by <i>trid</i> , if successful.
29894		
29895	ERRORS	
29896	TRL	The <i>posix_trace_create()</i> and <i>posix_trace_create_withlog()</i> functions shall fail if:
29897	[EAGAIN]	No more trace streams can be started now. {TRACE_SYS_MAX} has been exceeded.
29898		
29899	[EINTR]	The operation was interrupted by a signal. No trace stream was created.
29900	[EINVAL]	One or more of the trace parameters specified by the <i>attr</i> parameter is invalid.

29901	[ENOMEM]	The implementation does not currently have sufficient memory to create the trace stream with the specified parameters.
29902		
29903	[EPERM]	The caller does not have appropriate privilege to trace the process specified by <i>pid</i> .
29904		
29905	[ESRCH]	The <i>pid</i> argument does not refer to an existing process.
29906	TRL	The <i>posix_trace_create_withlog()</i> function shall fail if:
29907	[EBADF]	The <i>file_desc</i> argument is not a valid file descriptor open for writing.
29908	[EINVAL]	The <i>file_desc</i> argument refers to a file with a file type that does not support the log policy associated with the trace log.
29909		
29910	[ENOSPC]	No space left on device. The device corresponding to the argument <i>file_desc</i> does not contain the space required to create this trace log.
29911		
29912		
29913	TRL	The <i>posix_trace_flush()</i> and <i>posix_trace_shutdown()</i> functions shall fail if:
29914	[EINVAL]	The value of the <i>trid</i> argument does not correspond to an active trace stream with log.
29915		
29916	[EFBIG]	The trace log file has attempted to exceed an implementation-defined maximum file size.
29917		
29918	[ENOSPC]	No space left on device.
29919		
29920	EXAMPLES	
29921	None.	
29922	APPLICATION USAGE	
29923	None.	
29924	RATIONALE	
29925	None.	
29926	FUTURE DIRECTIONS	
29927	None.	
29928	SEE ALSO	
29929	<i>clock_getres()</i> , <i>exec</i> , <i>posix_trace_attr_init()</i> , <i>posix_trace_clear()</i> , <i>posix_trace_close()</i> ,	
29930	<i>posix_trace_eventid_equal()</i> , <i>posix_trace_eventtypelist_getnext_id()</i> , <i>posix_trace_flush()</i> ,	
29931	<i>posix_trace_get_attr()</i> , <i>posix_trace_get_filter()</i> , <i>posix_trace_get_status()</i> , <i>posix_trace_getnext_event()</i> ,	
29932	<i>posix_trace_open()</i> , <i>posix_trace_set_filter()</i> , <i>posix_trace_shutdown()</i> , <i>posix_trace_start()</i> ,	
29933	<i>posix_trace_timedgetnext_event()</i> , <i>posix_trace_trid_eventid_open()</i> , <i>posix_trace_start()</i> , <i>time()</i> , the	
29934	Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <trace.h>	
29935	CHANGE HISTORY	
29936	First released in Issue 6. Derived from IEEE Std 1003.1q-2000.	

29937 NAME

29938 `posix_trace_event`, `posix_trace_eventid_open` — trace functions for instrumenting application
29939 code (**TRACING**)

29940 SYNOPSIS

```
29941 TRC        #include <sys/types.h>
29942        #include <trace.h>

29943        void posix_trace_event(trace_event_id_t event_id,
29944                                const void *restrict data_ptr, size_t data_len);
29945        int posix_trace_eventid_open(const char *restrict event_name,
29946                                      trace_event_id_t *restrict event_id);
29947
```

29948 DESCRIPTION

29949 The `posix_trace_event()` function shall record the *event_id* and the user data pointed to by *data_ptr*
29950 in the trace stream into which the calling process is being traced and in which *event_id* is not
29951 filtered out. If the total size of the user trace event data represented by *data_len* is not greater
29952 than the declared maximum size for user trace event data, then the *truncation-status* attribute of
29953 the trace event recorded is `POSIX_TRACE_NOT_TRUNCATED`. Otherwise, the user trace event
29954 data is truncated to this declared maximum size and the *truncation-status* attribute of the trace
29955 event recorded is `POSIX_TRACE_TRUNCATED_RECORD`.

29956 If there is no trace stream created for the process or if the created trace stream is not running, or
29957 if the trace event specified by *event_id* is filtered out in the trace stream, the `posix_trace_event()`
29958 function shall have no effect.

29959 The `posix_trace_eventid_open()` function shall associate a user trace event name with a trace event
29960 type identifier for the calling process. The trace event name is the string pointed to by the
29961 argument *event_name*. It shall have a maximum of `{TRACE_EVENT_NAME_MAX}` characters
29962 (which has the minimum value `{POSIX_TRACE_EVENT_NAME_MAX}`). The number of user
29963 trace event type identifiers that can be defined for any given process is limited by the maximum
29964 value `{TRACE_USER_EVENT_MAX}`, which has the minimum value
29965 `{POSIX_TRACE_USER_EVENT_MAX}`.

29966 If the Trace Inherit option is not supported, the `posix_trace_eventid_open()` function shall
29967 associate the user trace event name pointed to by the *event_name* argument with a trace event
29968 type identifier that is unique for the traced process, and is returned in the variable pointed to by
29969 the *event_id* argument. If the user trace event name has already been mapped for the traced
29970 process, then the previously assigned trace event type identifier shall be returned. If the per-
29971 process user trace event name limit represented by `{TRACE_USER_EVENT_MAX}` has been
29972 reached, the pre-defined `POSIX_TRACE_UNNAMED_USEREVENT` (see Table 2-7 (on page 79))
29973 user trace event shall be returned.

29974 TRI If the Trace Inherit option is supported, the `posix_trace_eventid_open()` function shall associate the
29975 user trace event name pointed to by the *event_name* argument with a trace event type identifier
29976 that is unique for all the processes being traced in this same trace stream, and is returned in the
29977 variable pointed to by the *event_id* argument. If the user trace event name has already been
29978 mapped for the traced processes, then the previously assigned trace event type identifier shall be
29979 returned. If the per-process user trace event name limit represented by
29980 `{TRACE_USER_EVENT_MAX}` has been reached, the pre-defined
29981 `POSIX_TRACE_UNNAMED_USEREVENT` (Table 2-7 (on page 79)) user trace event shall be
29982 returned.

29983 **Note:** The above procedure, together with the fact that multiple processes can only be traced into the
29984 same trace stream by inheritance, ensure that all the processes that are traced into a trace
29985 stream have the same mapping of trace event names to trace event type identifiers.

29986

29987 If there is no trace stream created, the *posix_trace_eventid_open()* function shall store this
 29988 information for future trace streams created for this process.

29989 RETURN VALUE

29990 No return value is defined for the *posix_trace_event()* function.

29991 Upon successful completion, the *posix_trace_eventid_open()* function shall return a value of zero.
 29992 Otherwise, it shall return the corresponding error number. The *posix_trace_eventid_open()*
 29993 function stores the trace event type identifier value in the object pointed to by *event_id*, if
 29994 successful.

29995 ERRORS

29996 The *posix_trace_eventid_open()* function shall fail if:

29997 [ENAMETOOLONG]

29998 The size of the name pointed to by the *event_name* argument was longer than
 29999 the implementation-defined value {TRACE_EVENT_NAME_MAX}.

30000 EXAMPLES

30001 None.

30002 APPLICATION USAGE

30003 None.

30004 RATIONALE

30005 None.

30006 FUTURE DIRECTIONS

30007 None.

30008 SEE ALSO

30009 Table 2-7 (on page 79), *posix_trace_start()*, *posix_trace_trid_eventid_open()*, the Base Definitions
 30010 volume of IEEE Std 1003.1-2001, <sys/types.h>, <trace.h>

30011 CHANGE HISTORY

30012 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

30013 IEEE PASC Interpretation 1003.1 #123 is applied.

30014 IEEE PASC Interpretation 1003.1 #127 is applied, correcting some editorial errors in the names of
 30015 the *posix_trace_eventid_open()* function and the *event_id* argument.

30016 NAME

30017 `posix_trace_eventid_equal`, `posix_trace_eventid_get_name`, `posix_trace_trid_eventid_open` —
30018 manipulate the trace event type identifier (**TRACING**)

30019 SYNOPSIS

```
30020 TRC    #include <trace.h>

30021    int posix_trace_eventid_equal(trace_id_t trid, trace_event_id_t event1,
30022                                trace_event_id_t event2);
30023    int posix_trace_eventid_get_name(trace_id_t trid,
30024                                    trace_event_id_t event, char *event_name);
30025 TRC TEF  int posix_trace_trid_eventid_open(trace_id_t trid,
30026                                             const char *restrict event_name,
30027                                             trace_event_id_t *restrict event);
30028
```

30029 DESCRIPTION

30030 The `posix_trace_eventid_equal()` function shall compare the trace event type identifiers *event1* and
30031 *event2* from the same trace stream or the same trace log identified by the *trid* argument. If the
30032 trace event type identifiers *event1* and *event2* are from different trace streams, the return value
30033 shall be unspecified.

30034 The `posix_trace_eventid_get_name()` function shall return, in the argument pointed to by
30035 *event_name*, the trace event name associated with the trace event type identifier identified by the
30036 argument *event*, for the trace stream or for the trace log identified by the *trid* argument. The
30037 name of the trace event shall have a maximum of {TRACE_EVENT_NAME_MAX} characters
30038 (which has the minimum value {_POSIX_TRACE_EVENT_NAME_MAX}). Successive calls to
30039 this function with the same trace event type identifier and the same trace stream identifier shall
30040 return the same event name.

30041 TEF The `posix_trace_trid_eventid_open()` function shall associate a user trace event name with a trace
30042 event type identifier for a given trace stream. The trace stream is identified by the *trid* argument,
30043 and it shall be an active trace stream. The trace event name is the string pointed to by the
30044 argument *event_name*. It shall have a maximum of {TRACE_EVENT_NAME_MAX} characters
30045 (which has the minimum value {_POSIX_TRACE_EVENT_NAME_MAX}). The number of user
30046 trace event type identifiers that can be defined for any given process is limited by the maximum
30047 value {TRACE_USER_EVENT_MAX}, which has the minimum value
30048 {_POSIX_TRACE_USER_EVENT_MAX}.

30049 If the Trace Inherit option is not supported, the `posix_trace_trid_eventid_open()` function shall
30050 associate the user trace event name pointed to by the *event_name* argument with a trace event
30051 type identifier that is unique for the process being traced in the trace stream identified by the *trid*
30052 argument, and is returned in the variable pointed to by the *event* argument. If the user trace
30053 event name has already been mapped for the traced process, then the previously assigned trace
30054 event type identifier shall be returned. If the per-process user trace event name limit represented
30055 by {TRACE_USER_EVENT_MAX} has been reached, the pre-defined
30056 POSIX_TRACE_UNNAMED_USEREVENT (see Table 2-7 (on page 79)) user trace event shall be
30057 returned.

30058 TEF TRI If the Trace Inherit option is supported, the `posix_trace_trid_eventid_open()` function shall
30059 associate the user trace event name pointed to by the *event_name* argument with a trace event
30060 type identifier that is unique for all the processes being traced in the trace stream identified by
30061 the *trid* argument, and is returned in the variable pointed to by the *event* argument. If the user
30062 trace event name has already been mapped for the traced processes, then the previously
30063 assigned trace event type identifier shall be returned. If the per-process user trace event name
30064 limit represented by {TRACE_USER_EVENT_MAX} has been reached, the pre-defined

30065 POSIX_TRACE_UNNAMED_USEREVENT (see Table 2-7 (on page 79)) user trace event shall be
 30066 returned.

30067 RETURN VALUE

30068 TEF Upon successful completion, the *posix_trace_eventid_get_name()* and
 30069 *posix_trace_trid_eventid_open()* functions shall return a value of zero. Otherwise, they shall return
 30070 the corresponding error number.

30071 The *posix_trace_eventid_equal()* function shall return a non-zero value if *event1* and *event2* are
 30072 equal; otherwise, a value of zero shall be returned. No errors are defined. If either *event1* or
 30073 *event2* are not valid trace event type identifiers for the trace stream specified by *trid* or if the *trid*
 30074 is invalid, the behavior shall be unspecified.

30075 The *posix_trace_eventid_get_name()* function stores the trace event name value in the object
 30076 pointed to by *event_name*, if successful.

30077 TEF The *posix_trace_trid_eventid_open()* function stores the trace event type identifier value in the
 30078 object pointed to by *event*, if successful.

30079 ERRORS

30080 TEF The *posix_trace_eventid_get_name()* and *posix_trace_trid_eventid_open()* functions shall fail if:

30081 [EINVAL] The *trid* argument was not a valid trace stream identifier.

30082 TEF The *posix_trace_trid_eventid_open()* function shall fail if:

30083 TEF [ENAMETOOLONG]

30084 The size of the name pointed to by the *event_name* argument was longer than
 30085 the implementation-defined value {TRACE_EVENT_NAME_MAX}.

30086 The *posix_trace_eventid_get_name()* function shall fail if:

30087 [EINVAL] The trace event type identifier *event* was not associated with any name.

30088 EXAMPLES

30089 None.

30090 APPLICATION USAGE

30091 None.

30092 RATIONALE

30093 None.

30094 FUTURE DIRECTIONS

30095 None.

30096 SEE ALSO

30097 Table 2-7 (on page 79), *posix_trace_event()*, *posix_trace_getnext_event()*, the Base Definitions
 30098 volume of IEEE Std 1003.1-2001, <trace.h>

30099 CHANGE HISTORY

30100 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

30101 IEEE PASC Interpretations 1003.1 #123 and #129 are applied.

30102 NAME

30103 posix_trace_eventid_open — trace functions for instrumenting application code (**TRACING**)

30104 SYNOPSIS

30105 TRC #include <sys/types.h>

30106 #include <trace.h>

30107 int posix_trace_eventid_open(const char *restrict event_name,
30108 trace_event_id_t *restrict event_id);

30109

30110 DESCRIPTION

30111 Refer to *posix_trace_event()*.

30112 NAME

30113 posix_trace_eventset_add, posix_trace_eventset_del, posix_trace_eventset_empty,
 30114 posix_trace_eventset_fill, posix_trace_eventset_ismember — manipulate trace event type sets
 30115 (TRACING)

30116 SYNOPSIS

30117 TRC TEF #include <trace.h>

```
30118 int posix_trace_eventset_add(trace_event_id_t event_id,
30119     trace_event_set_t *set);
30120 int posix_trace_eventset_del(trace_event_id_t event_id,
30121     trace_event_set_t *set);
30122 int posix_trace_eventset_empty(trace_event_set_t *set);
30123 int posix_trace_eventset_fill(trace_event_set_t *set, int what);
30124 int posix_trace_eventset_ismember(trace_event_id_t event_id,
30125     const trace_event_set_t *restrict set, int *restrict ismember);
30126
```

30127 DESCRIPTION

30128 These primitives manipulate sets of trace event types. They operate on data objects addressable
 30129 by the application, not on the current trace event filter of any trace stream.

30130 The *posix_trace_eventset_add()* and *posix_trace_eventset_del()* functions, respectively, shall add or
 30131 delete the individual trace event type specified by the value of the argument *event_id* to or from
 30132 the trace event type set pointed to by the argument *set*. Adding a trace event type already in the
 30133 set or deleting a trace event type not in the set shall not be considered an error.

30134 The *posix_trace_eventset_empty()* function shall initialize the trace event type set pointed to by
 30135 the *set* argument such that all trace event types defined, both system and user, shall be excluded
 30136 from the set.

30137 The *posix_trace_eventset_fill()* function shall initialize the trace event type set pointed to by the
 30138 argument *set*, such that the set of trace event types defined by the argument *what* shall be
 30139 included in the set. The value of the argument *what* shall consist of one of the following values,
 30140 as defined in the <trace.h> header:

30141 POSIX_TRACE_WOPID_EVENTS

30142 All the process-independent implementation-defined system trace event types are included
 30143 in the set.

30144 POSIX_TRACE_SYSTEM_EVENTS

30145 All the implementation-defined system trace event types are included in the set, as are those
 30146 defined in IEEE Std 1003.1-2001.

30147 POSIX_TRACE_ALL_EVENTS

30148 All trace event types defined, both system and user, are included in the set.

30149 Applications shall call either *posix_trace_eventset_empty()* or *posix_trace_eventset_fill()* at least
 30150 once for each object of type **trace_event_set_t** prior to any other use of that object. If such an
 30151 object is not initialized in this way, but is nonetheless supplied as an argument to any of the
 30152 *posix_trace_eventset_add()*, *posix_trace_eventset_del()*, or *posix_trace_eventset_ismember()* functions,
 30153 the results are undefined.

30154 The *posix_trace_eventset_ismember()* function shall test whether the trace event type specified by
 30155 the value of the argument *event_id* is a member of the set pointed to by the argument *set*. The
 30156 value returned in the object pointed to by *ismember* argument is zero if the trace event type
 30157 identifier is not a member of the set and a value different from zero if it is a member of the set.

30158 RETURN VALUE

30159 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
30160 return the corresponding error number.

30161 ERRORS

30162 These functions may fail if:

30163 [EINVAL] The value of one of the arguments is invalid.

30164 EXAMPLES

30165 None.

30166 APPLICATION USAGE

30167 None.

30168 RATIONALE

30169 None.

30170 FUTURE DIRECTIONS

30171 None.

30172 SEE ALSO

30173 *posix_trace_set_filter()*, *posix_trace_trid_eventid_open()*, the Base Definitions volume of
30174 IEEE Std 1003.1-2001, <trace.h>

30175 CHANGE HISTORY

30176 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

30177 **NAME**

30178 `posix_trace_eventtypelist_getnext_id`, `posix_trace_eventtypelist_rewind` — iterate over a
 30179 mapping of trace event types (**TRACING**)

30180 **SYNOPSIS**

```
30181 TRC        #include <trace.h>

30182        int posix_trace_eventtypelist_getnext_id(trace_id_t trid,
30183                                                  trace_event_id_t *restrict event, int *restrict unavailable);
30184        int posix_trace_eventtypelist_rewind(trace_id_t trid);
30185
```

30186 **DESCRIPTION**

30187 The first time `posix_trace_eventtypelist_getnext_id()` is called, the function shall return in the
 30188 variable pointed to by *event* the first trace event type identifier of the list of trace events of the
 30189 trace stream identified by the *trid* argument. Successive calls to
 30190 `posix_trace_eventtypelist_getnext_id()` return in the variable pointed to by *event* the next trace
 30191 event type identifier in that same list. Each time a trace event type identifier is successfully
 30192 written into the variable pointed to by the *event* argument, the variable pointed to by the
 30193 *unavailable* argument shall be set to zero. When no more trace event type identifiers are
 30194 available, and so none is returned, the variable pointed to by the *unavailable* argument shall be
 30195 set to a value different from zero.

30196 The `posix_trace_eventtypelist_rewind()` function shall reset the next trace event type identifier to
 30197 be read to the first trace event type identifier from the list of trace events used in the trace stream
 30198 identified by *trid*.

30199 **RETURN VALUE**

30200 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
 30201 return the corresponding error number.

30202 The `posix_trace_eventtypelist_getnext_id()` function stores the trace event type identifier value in
 30203 the object pointed to by *event*, if successful.

30204 **ERRORS**

30205 These functions shall fail if:

30206 [EINVAL] The *trid* argument was not a valid trace stream identifier.

30207 **EXAMPLES**

30208 None.

30209 **APPLICATION USAGE**

30210 None.

30211 **RATIONALE**

30212 None.

30213 **FUTURE DIRECTIONS**

30214 None.

30215 **SEE ALSO**

30216 `posix_trace_event()`, `posix_trace_getnext_event()`, `posix_trace_trid_eventid_open()`, the Base
 30217 Definitions volume of IEEE Std 1003.1-2001, `<trace.h>`

30218 **CHANGE HISTORY**

30219 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

30220 IEEE PASC Interpretations 1003.1 #123 and #129 are applied.

30221 NAME

30222 posix_trace_flush — trace stream flush from a process (**TRACING**)

30223 SYNOPSIS

30224 TRC #include <sys/types.h>

30225 #include <trace.h>

30226 int posix_trace_flush(trace_id_t *trid*);

30227

30228 DESCRIPTION

30229 Refer to *posix_trace_create()*.

30230 **NAME**

30231 `posix_trace_get_attr`, `posix_trace_get_status` — retrieve the trace attributes or trace status
 30232 (**TRACING**)

30233 **SYNOPSIS**

30234 **TRC** `#include <trace.h>`

30235 `int posix_trace_get_attr(trace_id_t trid, trace_attr_t *attr);`

30236 `int posix_trace_get_status(trace_id_t trid,`

30237 `struct posix_trace_status_info *statusinfo);`

30238

30239 **DESCRIPTION**

30240 The `posix_trace_get_attr()` function shall copy the attributes of the active trace stream identified
 30241 **TRL** by *trid* into the object pointed to by the *attr* argument. If the Trace Log option is supported, *trid*
 30242 may represent a pre-recorded trace log.

30243 The `posix_trace_get_status()` function shall return, in the structure pointed to by the *statusinfo*
 30244 argument, the current trace status for the trace stream identified by the *trid* argument. These
 30245 status values returned in the structure pointed to by *statusinfo* shall have been appropriately
 30246 **TRL** read to ensure that the returned values are consistent. If the Trace Log option is supported and
 30247 the *trid* argument refers to a pre-recorded trace stream, the status shall be the status of the
 30248 completed trace stream.

30249 Each time the `posix_trace_get_status()` function is used, the overrun status of the trace stream
 30250 **TRL** shall be reset to `POSIX_TRACE_NO_OVERRUN` immediately after the call completes. If the
 30251 Trace Log option is supported, the `posix_trace_get_status()` function shall behave the same as
 30252 when the option is not supported except for the following differences:

30253 • If the *trid* argument refers to a trace stream with log, each time the `posix_trace_get_status()`
 30254 function is used, the log overrun status of the trace stream shall be reset to
 30255 `POSIX_TRACE_NO_OVERRUN` and the *flush_error* status shall be reset to zero immediately
 30256 after the call completes.

30257 • If the *trid* argument refers to a pre-recorded trace stream, the status returned shall be the
 30258 status of the completed trace stream and the status values of the trace stream shall not be
 30259 reset.
 30260

30261 **RETURN VALUE**

30262 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
 30263 return the corresponding error number.

30264 The `posix_trace_get_attr()` function stores the trace attributes in the object pointed to by *attr*, if
 30265 successful.

30266 The `posix_trace_get_status()` function stores the trace status in the object pointed to by *statusinfo*,
 30267 if successful.

30268 **ERRORS**

30269 These functions shall fail if:

30270 [**EINVAL**] The trace stream argument *trid* does not correspond to a valid active trace
 30271 stream or a valid trace log.

30272 EXAMPLES

30273 None.

30274 APPLICATION USAGE

30275 None.

30276 RATIONALE

30277 None.

30278 FUTURE DIRECTIONS

30279 None.

30280 SEE ALSO

30281 *posix_trace_attr_destroy()*, *posix_trace_attr_init()*, *posix_trace_create()*, *posix_trace_open()*, the Base

30282 Definitions volume of IEEE Std 1003.1-2001, <**trace.h**>

30283 CHANGE HISTORY

30284 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

30285 IEEE PASC Interpretation 1003.1 #123 is applied.

30286 **NAME**

30287 `posix_trace_get_filter`, `posix_trace_set_filter` — retrieve and set the filter of an initialized trace
 30288 stream (**TRACING**)

30289 **SYNOPSIS**

30290 TRC TEF `#include <trace.h>`

```
30291     int posix_trace_get_filter(trace_id_t trid, trace_event_set_t *set);
30292     int posix_trace_set_filter(trace_id_t trid,
30293                               const trace_event_set_t *set, int how);
30294
```

30295 **DESCRIPTION**

30296 The `posix_trace_get_filter()` function shall retrieve, into the argument pointed to by *set*, the actual
 30297 trace event filter from the trace stream specified by *trid*.

30298 The `posix_trace_set_filter()` function shall change the set of filtered trace event types after a trace
 30299 stream identified by the *trid* argument is created. This function may be called prior to starting
 30300 the trace stream, or while the trace stream is active. By default, if no call is made to
 30301 `posix_trace_set_filter()`, all trace events shall be recorded (that is, none of the trace event types are
 30302 filtered out).

30303 If this function is called while the trace is in progress, a special system trace event,
 30304 `POSIX_TRACE_FILTER`, shall be recorded in the trace indicating both the old and the new sets
 30305 of filtered trace event types (see Table 2-4 (on page 78) and Table 2-6 (on page 79)).

30306 If the `posix_trace_set_filter()` function is interrupted by a signal, an error shall be returned and the
 30307 filter shall not be changed. In this case, the state of the trace stream shall not be changed.

30308 The value of the argument *how* indicates the manner in which the set is to be changed and shall
 30309 have one of the following values, as defined in the `<trace.h>` header:

30310 `POSIX_TRACE_SET_EVENTSET`

30311 The resulting set of trace event types to be filtered shall be the trace event type set pointed
 30312 to by the argument *set*.

30313 `POSIX_TRACE_ADD_EVENTSET`

30314 The resulting set of trace event types to be filtered shall be the union of the current set and
 30315 the trace event type set pointed to by the argument *set*.

30316 `POSIX_TRACE_SUB_EVENTSET`

30317 The resulting set of trace event types to be filtered shall be all trace event types in the
 30318 current set that are not in the set pointed to by the argument *set*; that is, remove each
 30319 element of the specified set from the current filter.

30320 **RETURN VALUE**

30321 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
 30322 return the corresponding error number.

30323 The `posix_trace_get_filter()` function stores the set of filtered trace event types in *set*, if successful.

30324 **ERRORS**

30325 These functions shall fail if:

30326 `[EINVAL]` The value of the *trid* argument does not correspond to an active trace stream
 30327 or the value of the argument pointed to by *set* is invalid.

30328 `[EINTR]` The operation was interrupted by a signal.

30329 EXAMPLES

30330 None.

30331 APPLICATION USAGE

30332 None.

30333 RATIONALE

30334 None.

30335 FUTURE DIRECTIONS

30336 None.

30337 SEE ALSO

30338 Table 2-4 (on page 78), Table 2-6 (on page 79), *posix_trace_eventset_add()*, the Base Definitions
 30339 volume of IEEE Std 1003.1-2001, <**trace.h**>

30340 CHANGE HISTORY

30341 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

30342 IEEE PASC Interpretation 1003.1 #123 is applied.

30343 **NAME**30344 posix_trace_get_status — retrieve the trace status (**TRACING**)30345 **SYNOPSIS**

30346 TRC #include <trace.h>

30347 int posix_trace_get_status(trace_id_t *trid*,
30348 struct posix_trace_status_info **statusinfo*);

30349

30350 **DESCRIPTION**30351 Refer to *posix_trace_get_attr()*.

30352 NAME

30353 posix_trace_getnext_event, posix_trace_timedgetnext_event, posix_trace_trygetnext_event —
30354 retrieve a trace event (**TRACING**)

30355 SYNOPSIS

```
30356 TRC    #include <sys/types.h>
30357          #include <trace.h>

30358          int posix_trace_getnext_event(trace_id_t trid,
30359          struct posix_trace_event_info *restrict event,
30360          void *restrict data, size_t num_bytes,
30361          size_t *restrict data_len, int *restrict unavailable);
30362 TRC TMO int posix_trace_timedgetnext_event(trace_id_t trid,
30363          struct posix_trace_event_info *restrict event,
30364          void *restrict data, size_t num_bytes,
30365          size_t *restrict data_len, int *restrict unavailable,
30366          const struct timespec *restrict abs_timeout);
30367 TRC    int posix_trace_trygetnext_event(trace_id_t trid,
30368          struct posix_trace_event_info *restrict event,
30369          void *restrict data, size_t num_bytes,
30370          size_t *restrict data_len, int *restrict unavailable);
30371
```

30372 DESCRIPTION

30373 The *posix_trace_getnext_event()* function shall report a recorded trace event either from an active
30374 TRL trace stream without log or a pre-recorded trace stream identified by the *trid* argument. The
30375 *posix_trace_trygetnext_event()* function shall report a recorded trace event from an active trace
30376 stream without log identified by the *trid* argument.

30377 The trace event information associated with the recorded trace event shall be copied by the
30378 function into the structure pointed to by the argument *event* and the data associated with the
30379 trace event shall be copied into the buffer pointed to by the *data* argument.

30380 The *posix_trace_getnext_event()* function shall block if the *trid* argument identifies an active trace
30381 stream and there is currently no trace event ready to be retrieved. When returning, if a recorded
30382 trace event was reported, the variable pointed to by the *unavailable* argument shall be set to zero.
30383 Otherwise, the variable pointed to by the *unavailable* argument shall be set to a value different
30384 from zero.

30385 TMO The *posix_trace_timedgetnext_event()* function shall attempt to get another trace event from an
30386 active trace stream without log, as in the *posix_trace_getnext_event()* function. However, if no
30387 trace event is available from the trace stream, the implied wait shall be terminated when the
30388 timeout specified by the argument *abs_timeout* expires, and the function shall return the error
30389 [ETIMEDOUT].

30390 The timeout shall expire when the absolute time specified by *abs_timeout* passes, as measured by
30391 the clock upon which timeouts are based (that is, when the value of that clock equals or exceeds
30392 *abs_timeout*), or if the absolute time specified by *abs_timeout* has already passed at the time of the
30393 call.

30394 TMO TMR If the Timers option is supported, the timeout shall be based on the CLOCK_REALTIME clock;
30395 if the Timers option is not supported, the timeout shall be based on the system clock as returned
30396 by the *time()* function. The resolution of the timeout shall be the resolution of the clock on which
30397 it is based. The **timespec** data type is defined in the **<time.h>** header.

30398 TMO Under no circumstance shall the function fail with a timeout if a trace event is immediately
30399 available from the trace stream. The validity of the *abs_timeout* argument need not be checked if

30400 a trace event is immediately available from the trace stream.

30401 The behavior of this function for a pre-recorded trace stream is unspecified.

30402 TRL The *posix_trace_trygetnext_event()* function shall not block. This function shall return an error if
 30403 the *trid* argument identifies a pre-recorded trace stream. If a recorded trace event was reported,
 30404 the variable pointed to by the *unavailable* argument shall be set to zero. Otherwise, if no trace
 30405 event was reported, the variable pointed to by the *unavailable* argument shall be set to a value
 30406 different from zero.

30407 The argument *num_bytes* shall be the size of the buffer pointed to by the *data* argument. The
 30408 argument *data_len* reports to the application the length in bytes of the data record just
 30409 transferred. If *num_bytes* is greater than or equal to the size of the data associated with the trace
 30410 event pointed to by the *event* argument, all the recorded data shall be transferred. In this case, the
 30411 *truncation-status* member of the trace event structure shall be either
 30412 POSIX_TRACE_NOT_TRUNCATED, if the trace event data was recorded without truncation
 30413 while tracing, or POSIX_TRACE_TRUNCATED_RECORD, if the trace event data was truncated
 30414 when it was recorded. If the *num_bytes* argument is less than the length of recorded trace event
 30415 data, the data transferred shall be truncated to a length of *num_bytes*, the value stored in the
 30416 variable pointed to by *data_len* shall be equal to *num_bytes*, and the *truncation-status* member of
 30417 the *event* structure argument shall be set to POSIX_TRACE_TRUNCATED_READ (see the
 30418 **posix_trace_event_info** structure defined in <trace.h>).

30419 The report of a trace event shall be sequential starting from the oldest recorded trace event. Trace
 30420 events shall be reported in the order in which they were generated, up to an implementation-
 30421 defined time resolution that causes the ordering of trace events occurring very close to each
 30422 other to be unknown. Once reported, a trace event cannot be reported again from an active trace
 30423 stream. Once a trace event is reported from an active trace stream without log, the trace stream
 30424 shall make the resources associated with that trace event available to record future generated
 30425 trace events.

30426 **RETURN VALUE**

30427 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
 30428 return the corresponding error number.

30429 If successful, these functions store:

- 30430 • The recorded trace event in the object pointed to by *event*
- 30431 • The trace event information associated with the recorded trace event in the object pointed to
 30432 by *data*
- 30433 • The length of this trace event information in the object pointed to by *data_len*
- 30434 • The value of zero in the object pointed to by *unavailable*

30435 **ERRORS**

30436 These functions shall fail if:

30437 [EINVAL] The trace stream identifier argument *trid* is invalid.

30438 The *posix_trace_getnext_event()* and *posix_trace_timedgetnext_event()* functions shall fail if:

30439 [EINTR] The operation was interrupted by a signal, and so the call had no effect.

30440 The *posix_trace_trygetnext_event()* function shall fail if:

30441 [EINVAL] The trace stream identifier argument *trid* does not correspond to an active
 30442 trace stream.

30443	TMO	The <i>posix_trace_timedgetnext_event()</i> function shall fail if:
30444	[EINVAL]	There is no trace event immediately available from the trace stream, and the <i>timeout</i> argument is invalid.
30445		
30446	[ETIMEDOUT]	No trace event was available from the trace stream before the specified timeout <i>timeout</i> expired.
30447		
30448		

30449 EXAMPLES

30450 None.

30451 APPLICATION USAGE

30452 None.

30453 RATIONALE

30454 None.

30455 FUTURE DIRECTIONS

30456 None.

30457 SEE ALSO

30458 *posix_trace_create()*, *posix_trace_open()*, the Base Definitions volume of IEEE Std 1003.1-2001,
 30459 *<sys/types.h>*, *<trace.h>*

30460 CHANGE HISTORY

30461 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

30462 IEEE PASC Interpretation 1003.1 #123 is applied.

30463 **NAME**30464 posix_trace_open, posix_trace_rewind — trace log management (**TRACING**)30465 **SYNOPSIS**

30466 TCT TRL #include <trace.h>

30467 int posix_trace_open(int *file_desc*, trace_id_t **trid*);30468 int posix_trace_rewind(trace_id_t *trid*);

30469

30470 **DESCRIPTION**30471 Refer to *posix_trace_close()*.

30472 NAME

30473 posix_trace_set_filter — set filter of an initialized trace stream (**TRACING**)

30474 SYNOPSIS

30475 TRC TEF #include <trace.h>

```
30476       int posix_trace_set_filter(trace_id_t trid,  
30477           const trace_event_set_t *set, int how);
```

30478

30479 DESCRIPTION

30480 Refer to *posix_trace_get_filter()*.

30481 **NAME**30482 posix_trace_shutdown — trace stream shutdown from a process (**TRACING**)30483 **SYNOPSIS**

30484 TRC #include <sys/types.h>

30485 #include <trace.h>

30486 int posix_trace_shutdown(trace_id_t *trid*);

30487

30488 **DESCRIPTION**30489 Refer to *posix_trace_create()*.

30490 NAME

30491 `posix_trace_start`, `posix_trace_stop` — trace start and stop (**TRACING**)

30492 SYNOPSIS

```
30493 TRC    #include <trace.h>

30494        int posix_trace_start(trace_id_t trid);
30495        int posix_trace_stop (trace_id_t trid);
30496
```

30497 DESCRIPTION

30498 The `posix_trace_start()` and `posix_trace_stop()` functions, respectively, shall start and stop the
30499 trace stream identified by the argument *trid*.

30500 The effect of calling the `posix_trace_start()` function shall be recorded in the trace stream as the
30501 POSIX_TRACE_START system trace event and the status of the trace stream shall become
30502 POSIX_TRACE_RUNNING. If the trace stream is in progress when this function is called, the
30503 POSIX_TRACE_START system trace event shall not be recorded and the trace stream shall
30504 continue to run. If the trace stream is full, the POSIX_TRACE_START system trace event shall
30505 not be recorded and the status of the trace stream shall not be changed.

30506 The effect of calling the `posix_trace_stop()` function shall be recorded in the trace stream as the
30507 POSIX_TRACE_STOP system trace event and the status of the trace stream shall become
30508 POSIX_TRACE_SUSPENDED. If the trace stream is suspended when this function is called, the
30509 POSIX_TRACE_STOP system trace event shall not be recorded and the trace stream shall remain
30510 suspended. If the trace stream is full, the POSIX_TRACE_STOP system trace event shall not be
30511 recorded and the status of the trace stream shall not be changed.

30512 RETURN VALUE

30513 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
30514 return the corresponding error number.

30515 ERRORS

30516 These functions shall fail if:

30517 [EINVAL] The value of the argument *trid* does not correspond to an active trace stream
30518 and thus no trace stream was started or stopped.

30519 [EINTR] The operation was interrupted by a signal and thus the trace stream was not
30520 necessarily started or stopped.

30521 EXAMPLES

30522 None.

30523 APPLICATION USAGE

30524 None.

30525 RATIONALE

30526 None.

30527 FUTURE DIRECTIONS

30528 None.

30529 SEE ALSO

30530 `posix_trace_create()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<trace.h>`

30531 CHANGE HISTORY

- 30532 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.
- 30533 IEEE PASC Interpretation 1003.1 #123 is applied.

30534 NAME

30535 posix_trace_timedgetnext_event, — retrieve a trace event (**TRACING**)

30536 SYNOPSIS

30537 TRC TMO #include <sys/types.h>

30538 #include <trace.h>

```
30539       int posix_trace_timedgetnext_event(trace_id_t trid,  
30540       struct posix_trace_event_info *restrict event,  
30541       void *restrict data, size_t num_bytes,  
30542       size_t *restrict data_len, int *restrict unavailable,  
30543       const struct timespec *restrict abs_timeout);  
30544
```

30545 DESCRIPTION

30546 Refer to *posix_trace_getnext_event()*.

30547 **NAME**30548 `posix_trace_trid_eventid_open` — open a trace event type identifier (**TRACING**)30549 **SYNOPSIS**30550 TRC TEF `#include <trace.h>`

```
30551        int posix_trace_trid_eventid_open(trace_id_t trid,  
30552                                        const char *restrict event_name,  
30553                                        trace_event_id_t *restrict event);
```

30554

30555 **DESCRIPTION**30556 Refer to `posix_trace_eventid_equal()`.

30557 NAME

30558 `posix_trace_trygetnext_event` — retrieve a trace event (**TRACING**)

30559 SYNOPSIS

30560 TRC `#include <sys/types.h>`

30561 `#include <trace.h>`

```
30562        int posix_trace_trygetnext_event(trace_id_t trid,  
30563                 struct posix_trace_event_info *restrict event,  
30564                 void *restrict data, size_t num_bytes,  
30565                 size_t *restrict data_len, int *restrict unavailable);  
30566
```

30567 DESCRIPTION

30568 Refer to `posix_trace_getnext_event()`.

30569 NAME

30570 `posix_typed_mem_get_info` — query typed memory information (**ADVANCED REALTIME**)

30571 SYNOPSIS

30572 **TYM** `#include <sys/mman.h>`

```
30573 int posix_typed_mem_get_info(int fildes,
30574 struct posix_typed_mem_info *info);
```

30575

30576 DESCRIPTION

30577 The `posix_typed_mem_get_info()` function shall return, in the `posix_tmi_length` field of the
 30578 **posix_typed_mem_info** structure pointed to by *info*, the maximum length which may be
 30579 successfully allocated by the typed memory object designated by *fildes*. This maximum length
 30580 shall take into account the flag `POSIX_TYPED_MEM_ALLOCATE` or
 30581 `POSIX_TYPED_MEM_ALLOCATE_CONTIG` specified when the typed memory object
 30582 represented by *fildes* was opened. The maximum length is dynamic; therefore, the value returned
 30583 is valid only while the current mapping of the corresponding typed memory pool remains
 30584 unchanged.

30585 If *fildes* represents a typed memory object opened with neither the
 30586 `POSIX_TYPED_MEM_ALLOCATE` flag nor the `POSIX_TYPED_MEM_ALLOCATE_CONTIG`
 30587 flag specified, the returned value of *info*->`posix_tmi_length` is unspecified.

30588 The `posix_typed_mem_get_info()` function may return additional implementation-defined
 30589 information in other fields of the **posix_typed_mem_info** structure pointed to by *info*.

30590 If the memory object specified by *fildes* is not a typed memory object, then the behavior of this
 30591 function is undefined.

30592 RETURN VALUE

30593 Upon successful completion, the `posix_typed_mem_get_info()` function shall return zero;
 30594 otherwise, the corresponding error status value shall be returned.

30595 ERRORS

30596 The `posix_typed_mem_get_info()` function shall fail if:

30597 `[EBADF]` The *fildes* argument is not a valid open file descriptor.

30598 `[ENODEV]` The *fildes* argument is not connected to a memory object supported by this
 30599 function.

30600 This function shall not return an error code of `[EINTR]`.

30601 EXAMPLES

30602 None.

30603 APPLICATION USAGE

30604 None.

30605 RATIONALE

30606 An application that needs to allocate a block of typed memory with length dependent upon the
 30607 amount of memory currently available must either query the typed memory object to obtain the
 30608 amount available, or repeatedly invoke `mmap()` attempting to guess an appropriate length.
 30609 While the latter method is existing practice with `malloc()`, it is awkward and imprecise. The
 30610 `posix_typed_mem_get_info()` function allows an application to immediately determine available
 30611 memory. This is particularly important for typed memory objects that may in some cases be
 30612 scarce resources. Note that when a typed memory pool is a shared resource, some form of
 30613 mutual-exclusion or synchronization may be required while typed memory is being queried and

30614 allocated to prevent race conditions.

30615 The existing *fstat()* function is not suitable for this purpose. We realize that implementations
30616 may wish to provide other attributes of typed memory objects (for example, alignment
30617 requirements, page size, and so on). The *fstat()* function returns a structure which is not
30618 extensible and, furthermore, contains substantial information that is inappropriate for typed
30619 memory objects.

30620 **FUTURE DIRECTIONS**

30621 None.

30622 **SEE ALSO**

30623 *fstat()*, *mmap()*, *posix_typed_mem_open()*, the Base Definitions volume of IEEE Std 1003.1-2001,
30624 **<sys/mman.h>**

30625 **CHANGE HISTORY**

30626 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

30627 **NAME**30628 posix_typed_mem_open — open a typed memory object (**ADVANCED REALTIME**)30629 **SYNOPSIS**30630 **TYM** `#include <sys/mman.h>`30631 `int posix_typed_mem_open(const char *name, int oflag, int tflag);`

30632

30633 **DESCRIPTION**

30634 The *posix_typed_mem_open()* function shall establish a connection between the typed memory
 30635 object specified by the string pointed to by *name* and a file descriptor. It shall create an open file
 30636 description that refers to the typed memory object and a file descriptor that refers to that open
 30637 file description. The file descriptor is used by other functions to refer to that typed memory
 30638 object. It is unspecified whether the name appears in the file system and is visible to other
 30639 functions that take pathnames as arguments. The *name* argument shall conform to the
 30640 construction rules for a pathname. If *name* begins with the slash character, then processes calling
 30641 *posix_typed_mem_open()* with the same value of *name* shall refer to the same typed memory
 30642 object. If *name* does not begin with the slash character, the effect is implementation-defined. The
 30643 interpretation of slash characters other than the leading slash character in *name* is
 30644 implementation-defined.

30645 Each typed memory object supported in a system shall be identified by a name which specifies
 30646 not only its associated typed memory pool, but also the path or port by which it is accessed. That
 30647 is, the same typed memory pool accessed via several different ports shall have several different
 30648 corresponding names. The binding between names and typed memory objects is established in
 30649 an implementation-defined manner. Unlike shared memory objects, there is no way within
 30650 IEEE Std 1003.1-2001 for a program to create a typed memory object.

30651 The value of *tflag* shall determine how the typed memory object behaves when subsequently
 30652 mapped by calls to *mmap()*. At most, one of the following flags defined in `<sys/mman.h>` may
 30653 be specified:

30654 **POSIX_TYPED_MEM_ALLOCATE**30655 Allocate on *mmap()*.30656 **POSIX_TYPED_MEM_ALLOCATE_CONTIG**30657 Allocate contiguously on *mmap()*.30658 **POSIX_TYPED_MEM_MAP_ALLOCATABLE**30659 Map on *mmap()*, without affecting allocatability.

30660 If *tflag* has the flag **POSIX_TYPED_MEM_ALLOCATE** specified, any subsequent call to *mmap()*
 30661 using the returned file descriptor shall result in allocation and mapping of typed memory from
 30662 the specified typed memory pool. The allocated memory may be a contiguous previously
 30663 unallocated area of the typed memory pool or several non-contiguous previously unallocated
 30664 areas (mapped to a contiguous portion of the process address space). If *tflag* has the flag
 30665 **POSIX_TYPED_MEM_ALLOCATE_CONTIG** specified, any subsequent call to *mmap()* using the
 30666 returned file descriptor shall result in allocation and mapping of a single contiguous previously
 30667 unallocated area of the typed memory pool (also mapped to a contiguous portion of the process
 30668 address space). If *tflag* has none of the flags **POSIX_TYPED_MEM_ALLOCATE** or
 30669 **POSIX_TYPED_MEM_ALLOCATE_CONTIG** specified, any subsequent call to *mmap()* using the
 30670 returned file descriptor shall map an application-chosen area from the specified typed memory
 30671 pool such that this mapped area becomes unavailable for allocation until unmapped by all
 30672 processes. If *tflag* has the flag **POSIX_TYPED_MEM_MAP_ALLOCATABLE** specified, any
 30673 subsequent call to *mmap()* using the returned file descriptor shall map an application-chosen
 30674 area from the specified typed memory pool without an effect on the availability of that area for

allocation; that is, mapping such an object leaves each byte of the mapped area unallocated if it was unallocated prior to the mapping or allocated if it was allocated prior to the mapping. The appropriate privilege to specify the `POSIX_TYPED_MEM_MAP_ALLOCATABLE` flag is implementation-defined.

If successful, *posix_typed_mem_open()* shall return a file descriptor for the typed memory object that is the lowest numbered file descriptor not currently open for that process. The open file description is new, and therefore the file descriptor shall not share it with any other processes. It is unspecified whether the file offset is set. The `FD_CLOEXEC` file descriptor flag associated with the new file descriptor shall be cleared.

The behavior of *msync()*, *ftruncate()*, and all file operations other than *mmap()*, *posix_mem_offset()*, *posix_typed_mem_get_info()*, *fstat()*, *dup()*, *dup2()*, and *close()*, is unspecified when passed a file descriptor connected to a typed memory object by this function.

The file status flags of the open file description shall be set according to the value of *oflag*. Applications shall specify exactly one of the three access mode values described below and defined in the `<fcntl.h>` header, as the value of *oflag*.

30690	<code>O_RDONLY</code>	Open for read access only.
30691	<code>O_WRONLY</code>	Open for write access only.
30692	<code>O_RDWR</code>	Open for read or write access.

RETURN VALUE

Upon successful completion, the *posix_typed_mem_open()* function shall return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, it shall return `-1` and set *errno* to indicate the error.

ERRORS

The *posix_typed_mem_open()* function shall fail if:

30699 30700	<code>[EACCES]</code>	The typed memory object exists and the permissions specified by <i>oflag</i> are denied.
30701	<code>[EINTR]</code>	The <i>posix_typed_mem_open()</i> operation was interrupted by a signal.
30702 30703 30704 30705	<code>[EINVAL]</code>	The flags specified in <i>tflag</i> are invalid (more than one of <code>POSIX_TYPED_MEM_ALLOCATE</code> , <code>POSIX_TYPED_MEM_ALLOCATE_CONTIG</code> , or <code>POSIX_TYPED_MEM_MAP_ALLOCATABLE</code> is specified).
30706	<code>[EMFILE]</code>	Too many file descriptors are currently in use by this process.
30707 30708 30709	<code>[ENAMETOOLONG]</code>	The length of the <i>name</i> argument exceeds <code>{PATH_MAX}</code> or a pathname component is longer than <code>{NAME_MAX}</code> .
30710	<code>[ENFILE]</code>	Too many file descriptors are currently open in the system.
30711	<code>[ENOENT]</code>	The named typed memory object does not exist.
30712 30713	<code>[EPERM]</code>	The caller lacks the appropriate privilege to specify the flag <code>POSIX_TYPED_MEM_MAP_ALLOCATABLE</code> in argument <i>tflag</i> .

30714 **EXAMPLES**

30715 None.

30716 **APPLICATION USAGE**

30717 None.

30718 **RATIONALE**

30719 None.

30720 **FUTURE DIRECTIONS**

30721 None.

30722 **SEE ALSO**

30723 *close()*, *dup()*, *exec*, *fcntl()*, *fstat()*, *ftruncate()*, *mmap()*, *msync()*, *posix_mem_offset()*,
30724 *posix_typed_mem_get_info()*, *umask()*, the Base Definitions volume of IEEE Std 1003.1-2001,
30725 <fcntl.h>, <sys/mman.h>

30726 **CHANGE HISTORY**

30727 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

30728 **NAME**

30729 pow, powf, powl — power function

30730 **SYNOPSIS**

30731 #include <math.h>

30732 double pow(double x, double y);

30733 float powf(float x, float y);

30734 long double powl(long double x, long double y);

30735 **DESCRIPTION**

30736 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 30737 conflict between the requirements described here and the ISO C standard is unintentional. This
 30738 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

30739 These functions shall compute the value of x raised to the power y , x^y . If x is negative, the
 30740 application shall ensure that y is an integer value.

30741 An application wishing to check for error situations should set *errno* to zero and call
 30742 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 30743 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 30744 zero, an error has occurred.

30745 **RETURN VALUE**30746 Upon successful completion, these functions shall return the value of x raised to the power y .

30747 **MX** For finite values of $x < 0$, and finite non-integer values of y , a domain error shall occur and either
 30748 a NaN (if representable), or an implementation-defined value shall be returned.

30749 If the correct value would cause overflow, a range error shall occur and *pow()*, *powf()*, and
 30750 *powl()* shall return HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

30751 If the correct value would cause underflow, and is not representable, a range error may occur,
 30752 **MX** and either 0.0 (if supported), or an implementation-defined value shall be returned.

30753 **MX** If x or y is a NaN, a NaN shall be returned (unless specified elsewhere in this description).30754 For any value of y (including NaN), if x is +1, 1.0 shall be returned.30755 For any value of x (including NaN), if y is ± 0 , 1.0 shall be returned.30756 For any odd integer value of $y > 0$, if x is ± 0 , ± 0 shall be returned.30757 For $y > 0$ and not an odd integer, if x is ± 0 , +0 shall be returned.30758 If x is -1 , and y is $\pm \text{Inf}$, 1.0 shall be returned.30759 For $|x| < 1$, if y is $-\text{Inf}$, +Inf shall be returned.30760 For $|x| > 1$, if y is $-\text{Inf}$, +0 shall be returned.30761 For $|x| < 1$, if y is +Inf, +0 shall be returned.30762 For $|x| > 1$, if y is +Inf, +Inf shall be returned.30763 For y an odd integer < 0 , if x is $-\text{Inf}$, -0 shall be returned.30764 For $y < 0$ and not an odd integer, if x is $-\text{Inf}$, +0 shall be returned.30765 For y an odd integer > 0 , if x is $-\text{Inf}$, $-\text{Inf}$ shall be returned.30766 For $y > 0$ and not an odd integer, if x is $-\text{Inf}$, +Inf shall be returned.

30767 For $y < 0$, if x is $+\text{Inf}$, $+0$ shall be returned.

30768 For $y > 0$, if x is $+\text{Inf}$, $+\text{Inf}$ shall be returned.

30769 For y an odd integer < 0 , if x is ± 0 , a pole error shall occur and $\pm\text{HUGE_VAL}$, $\pm\text{HUGE_VALF}$, and
 30770 $\pm\text{HUGE_VALL}$ shall be returned for $\text{pow}()$, $\text{powf}()$, and $\text{powl}()$, respectively.

30771 For $y < 0$ and not an odd integer, if x is ± 0 , a pole error shall occur and HUGE_VAL ,
 30772 HUGE_VALF , and HUGE_VALL shall be returned for $\text{pow}()$, $\text{powf}()$, and $\text{powl}()$, respectively.

30773 If the correct value would cause underflow, and is representable, a range error may occur and
 30774 the correct value shall be returned.

30775 ERRORS

30776 These functions shall fail if:

30777 Domain Error The value of x is negative and y is a finite non-integer.

30778 If the integer expression $(\text{math_errhandling} \ \& \ \text{MATH_ERRNO})$ is non-zero,
 30779 then *errno* shall be set to $[\text{EDOM}]$. If the integer expression $(\text{math_errhandling}$
 30780 $\ \& \ \text{MATH_ERREXCEPT})$ is non-zero, then the invalid floating-point exception
 30781 shall be raised.

30782 MX Pole Error The value of x is zero and y is negative.

30783 If the integer expression $(\text{math_errhandling} \ \& \ \text{MATH_ERRNO})$ is non-zero,
 30784 then *errno* shall be set to $[\text{ERANGE}]$. If the integer expression
 30785 $(\text{math_errhandling} \ \& \ \text{MATH_ERREXCEPT})$ is non-zero, then the divide-by-
 30786 zero floating-point exception shall be raised.

30787 Range Error The result overflows.

30788 If the integer expression $(\text{math_errhandling} \ \& \ \text{MATH_ERRNO})$ is non-zero,
 30789 then *errno* shall be set to $[\text{ERANGE}]$. If the integer expression
 30790 $(\text{math_errhandling} \ \& \ \text{MATH_ERREXCEPT})$ is non-zero, then the overflow
 30791 floating-point exception shall be raised.

30792 These functions may fail if:

30793 Range Error The result underflows.

30794 If the integer expression $(\text{math_errhandling} \ \& \ \text{MATH_ERRNO})$ is non-zero,
 30795 then *errno* shall be set to $[\text{ERANGE}]$. If the integer expression
 30796 $(\text{math_errhandling} \ \& \ \text{MATH_ERREXCEPT})$ is non-zero, then the underflow
 30797 floating-point exception shall be raised.

30798 EXAMPLES

30799 None.

30800 APPLICATION USAGE

30801 On error, the expressions $(\text{math_errhandling} \ \& \ \text{MATH_ERRNO})$ and $(\text{math_errhandling} \ \& \ \text{MATH_ERREXCEPT})$
 30802 are independent of each other, but at least one of them must be non-zero.

30803 RATIONALE

30804 None.

30805 FUTURE DIRECTIONS

30806 None.

30807 **SEE ALSO**

30808 *exp()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-2001,
30809 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

30810 **CHANGE HISTORY**

30811 First released in Issue 1. Derived from Issue 1 of the SVID.

30812 **Issue 5**

30813 The DESCRIPTION is updated to indicate how an application should check for an error. This
30814 text was previously published in the APPLICATION USAGE section.

30815 **Issue 6**

30816 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

30817 The *powf()* and *powl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

30818 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
30819 revised to align with the ISO/IEC 9899:1999 standard.

30820 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
30821 marked.

30822 **NAME**30823

```
pread
```

 — read from a file30824 **SYNOPSIS**30825 XSI

```
#include <unistd.h>
```

30826

```
ssize_t pread(int fildev, void *buf, size_t nbyte, off_t offset);
```

30827

30828 **DESCRIPTION**30829 Refer to *read()*.

30830 NAME

30831 printf — print formatted output

30832 SYNOPSIS

30833 #include <stdio.h>

30834 int printf(const char *restrict *format*, ...);

30835 DESCRIPTION

30836 Refer to *fprintf()*.

30837 NAME

30838 pselect, select — synchronous I/O multiplexing

30839 SYNOPSIS

```

30840 #include <sys/select.h>

30841 int pselect(int nfds, fd_set *restrict readfds,
30842            fd_set *restrict writefds, fd_set *restrict errorfds,
30843            const struct timespec *restrict timeout,
30844            const sigset_t *restrict sigmask);
30845 int select(int nfds, fd_set *restrict readfds,
30846           fd_set *restrict writefds, fd_set *restrict errorfds,
30847           struct timeval *restrict timeout);
30848 void FD_CLR(int fd, fd_set *fdset);
30849 void FD_ISSET(int fd, fd_set *fdset);
30850 void FD_SET(int fd, fd_set *fdset);
30851 void FD_ZERO(fd_set *fdset);

```

30852 DESCRIPTION

30853 The *pselect()* function shall examine the file descriptor sets whose addresses are passed in the
 30854 *readfds*, *writefds*, and *errorfds* parameters to see whether some of their descriptors are ready for
 30855 reading, are ready for writing, or have an exceptional condition pending, respectively.

30856 The *select()* function shall be equivalent to the *pselect()* function, except as follows:

- 30857 • For the *select()* function, the timeout period is given in seconds and microseconds in an
 30858 argument of type **struct timeval**, whereas for the *pselect()* function the timeout period is
 30859 given in seconds and nanoseconds in an argument of type **struct timespec**.
- 30860 • The *select()* function has no *sigmask* argument; it shall behave as *pselect()* does when *sigmask*
 30861 is a null pointer.
- 30862 • Upon successful completion, the *select()* function may modify the object pointed to by the
 30863 *timeout* argument.

30864 The *pselect()* and *select()* functions shall support regular files, terminal and pseudo-terminal
 30865 XSR devices, **STREAMS**-based files, FIFOs, pipes, and sockets. The behavior of *pselect()* and *select()*
 30866 on file descriptors that refer to other types of file is unspecified.

30867 The *nfds* argument specifies the range of descriptors to be tested. The first *nfds* descriptors shall
 30868 be checked in each set; that is, the descriptors from zero through *nfds*–1 in the descriptor sets
 30869 shall be examined.

30870 If the *readfds* argument is not a null pointer, it points to an object of type **fd_set** that on input
 30871 specifies the file descriptors to be checked for being ready to read, and on output indicates
 30872 which file descriptors are ready to read.

30873 If the *writefds* argument is not a null pointer, it points to an object of type **fd_set** that on input
 30874 specifies the file descriptors to be checked for being ready to write, and on output indicates
 30875 which file descriptors are ready to write.

30876 If the *errorfds* argument is not a null pointer, it points to an object of type **fd_set** that on input
 30877 specifies the file descriptors to be checked for error conditions pending, and on output indicates
 30878 which file descriptors have error conditions pending.

30879 Upon successful completion, the *pselect()* or *select()* function shall modify the objects pointed to
 30880 by the *readfds*, *writefds*, and *errorfds* arguments to indicate which file descriptors are ready for
 30881 reading, ready for writing, or have an error condition pending, respectively, and shall return the
 30882 total number of ready descriptors in all the output sets. For each file descriptor less than *nfds*, the

30883 corresponding bit shall be set on successful completion if it was set on input and the associated
30884 condition is true for that file descriptor.

30885 If none of the selected descriptors are ready for the requested operation, the *pselect()* or *select()*
30886 function shall block until at least one of the requested operations becomes ready, until the
30887 *timeout* occurs, or until interrupted by a signal. The *timeout* parameter controls how long the
30888 *pselect()* or *select()* function shall take before timing out. If the *timeout* parameter is not a null
30889 pointer, it specifies a maximum interval to wait for the selection to complete. If the specified
30890 time interval expires without any requested operation becoming ready, the function shall return.
30891 If the *timeout* parameter is a null pointer, then the call to *pselect()* or *select()* shall block
30892 indefinitely until at least one descriptor meets the specified criteria. To effect a poll, the *timeout*
30893 parameter should not be a null pointer, and should point to a zero-valued **timespec** structure.

30894 The use of a timeout does not affect any pending timers set up by *alarm()*, *ualarm()*, or
30895 *setitimer()*.

30896 Implementations may place limitations on the maximum timeout interval supported. All
30897 implementations shall support a maximum timeout interval of at least 31 days. If the *timeout*
30898 argument specifies a timeout interval greater than the implementation-defined maximum value,
30899 the maximum value shall be used as the actual timeout value. Implementations may also place
30900 limitations on the granularity of timeout intervals. If the requested timeout interval requires a
30901 finer granularity than the implementation supports, the actual timeout interval shall be rounded
30902 up to the next supported value.

30903 If *sigmask* is not a null pointer, then the *pselect()* function shall replace the signal mask of the
30904 process by the set of signals pointed to by *sigmask* before examining the descriptors, and shall
30905 restore the signal mask of the process before returning.

30906 A descriptor shall be considered ready for reading when a call to an input function with
30907 O_NONBLOCK clear would not block, whether or not the function would transfer data
30908 successfully. (The function might return data, an end-of-file indication, or an error other than
30909 one indicating that it is blocked, and in each of these cases the descriptor shall be considered
30910 ready for reading.)

30911 A descriptor shall be considered ready for writing when a call to an output function with
30912 O_NONBLOCK clear would not block, whether or not the function would transfer data
30913 successfully.

30914 If a socket has a pending error, it shall be considered to have an exceptional condition pending.
30915 Otherwise, what constitutes an exceptional condition is file type-specific. For a file descriptor for
30916 use with a socket, it is protocol-specific except as noted below. For other file types it is
30917 implementation-defined. If the operation is meaningless for a particular file type, *pselect()* or
30918 *select()* shall indicate that the descriptor is ready for read or write operations, and shall indicate
30919 that the descriptor has no exceptional condition pending.

30920 If a descriptor refers to a socket, the implied input function is the *recvmsg()* function with
30921 parameters requesting normal and ancillary data, such that the presence of either type shall
30922 cause the socket to be marked as readable. The presence of out-of-band data shall be checked if
30923 the socket option SO_OOBINLINE has been enabled, as out-of-band data is enqueued with
30924 normal data. If the socket is currently listening, then it shall be marked as readable if an
30925 incoming connection request has been received, and a call to the *accept()* function shall complete
30926 without blocking.

30927 If a descriptor refers to a socket, the implied output function is the *sendmsg()* function supplying
30928 an amount of normal data equal to the current value of the SO_SNDLOWAT option for the
30929 socket. If a non-blocking call to the *connect()* function has been made for a socket, and the
30930 connection attempt has either succeeded or failed leaving a pending error, the socket shall be

30931 marked as writable.

30932 A socket shall be considered to have an exceptional condition pending if a receive operation
 30933 with `O_NONBLOCK` clear for the open file description and with the `MSG_OOB` flag set would
 30934 return out-of-band data without blocking. (It is protocol-specific whether the `MSG_OOB` flag
 30935 would be used to read out-of-band data.) A socket shall also be considered to have an
 30936 exceptional condition pending if an out-of-band data mark is present in the receive queue. Other
 30937 circumstances under which a socket may be considered to have an exceptional condition
 30938 pending are protocol-specific and implementation-defined.

30939 If the *readfds*, *writefds*, and *errorfds* arguments are all null pointers and the *timeout* argument is not
 30940 a null pointer, the *pselect()* or *select()* function shall block for the time specified, or until
 30941 interrupted by a signal. If the *readfds*, *writefds*, and *errorfds* arguments are all null pointers and the
 30942 *timeout* argument is a null pointer, the *pselect()* or *select()* function shall block until interrupted
 30943 by a signal.

30944 File descriptors associated with regular files shall always select true for ready to read, ready to
 30945 write, and error conditions.

30946 On failure, the objects pointed to by the *readfds*, *writefds*, and *errorfds* arguments shall not be
 30947 modified. If the timeout interval expires without the specified condition being true for any of the
 30948 specified file descriptors, the objects pointed to by the *readfds*, *writefds*, and *errorfds* arguments
 30949 shall have all bits set to 0.

30950 File descriptor masks of type *fd_set* can be initialized and tested with *FD_CLR()*, *FD_ISSET()*,
 30951 *FD_SET()*, and *FD_ZERO()*. It is unspecified whether each of these is a macro or a function. If a
 30952 macro definition is suppressed in order to access an actual function, or a program defines an
 30953 external identifier with any of these names, the behavior is undefined.

30954 *FD_CLR(fd, fdsetp)* shall remove the file descriptor *fd* from the set pointed to by *fdsetp*. If *fd* is not
 30955 a member of this set, there shall be no effect on the set, nor will an error be returned.

30956 *FD_ISSET(fd, fdsetp)* shall evaluate to non-zero if the file descriptor *fd* is a member of the set
 30957 pointed to by *fdsetp*, and shall evaluate to zero otherwise.

30958 *FD_SET(fd, fdsetp)* shall add the file descriptor *fd* to the set pointed to by *fdsetp*. If the file
 30959 descriptor *fd* is already in this set, there shall be no effect on the set, nor will an error be returned.

30960 *FD_ZERO(fdsetp)* shall initialize the descriptor set pointed to by *fdsetp* to the null set. No error is
 30961 returned if the set is not empty at the time *FD_ZERO()* is invoked.

30962 The behavior of these macros is undefined if the *fd* argument is less than 0 or greater than or
 30963 equal to `FD_SETSIZE`, or if *fd* is not a valid file descriptor, or if any of the arguments are
 30964 expressions with side effects.

30965 RETURN VALUE

30966 Upon successful completion, the *pselect()* and *select()* functions shall return the total number of
 30967 bits set in the bit masks. Otherwise, `-1` shall be returned, and *errno* shall be set to indicate the
 30968 error.

30969 *FD_CLR()*, *FD_SET()*, and *FD_ZERO()* do not return a value. *FD_ISSET()* shall return a non-
 30970 zero value if the bit for the file descriptor *fd* is set in the file descriptor set pointed to by *fdset*, and
 30971 0 otherwise.

30972 ERRORS

30973 Under the following conditions, *pselect()* and *select()* shall fail and set *errno* to:

30974 [EBADF] One or more of the file descriptor sets specified a file descriptor that is not a
 30975 valid open file descriptor.

30976	[EINTR]	The function was interrupted before any of the selected events occurred and
30977		before the timeout interval expired.
30978 XSI		If SA_RESTART has been set for the interrupting signal, it is implementation-
30979		defined whether the function restarts or returns with [EINTR].
30980	[EINVAL]	An invalid timeout interval was specified.
30981	[EINVAL]	The <i>nfds</i> argument is less than 0 or greater than FD_SETSIZE.
30982 XSR	[EINVAL]	One of the specified file descriptors refers to a STREAM or multiplexer that is
30983		linked (directly or indirectly) downstream from a multiplexer.

30984 EXAMPLES

30985 None.

30986 APPLICATION USAGE

30987 None.

30988 RATIONALE

30989 In previous versions of the Single UNIX Specification, the *select()* function was defined in the
 30990 `<sys/time.h>` header. This is now changed to `<sys/select.h>`. The rationale for this change was
 30991 as follows: the introduction of the *pselect()* function included the `<sys/select.h>` header and the
 30992 `<sys/select.h>` header defines all the related definitions for the *pselect()* and *select()* functions.
 30993 Backwards-compatibility to existing XSI implementations is handled by allowing `<sys/time.h>`
 30994 to include `<sys/select.h>`.

30995 FUTURE DIRECTIONS

30996 None.

30997 SEE ALSO

30998 *accept()*, *alarm()*, *connect()*, *fcntl()*, *poll()*, *read()*, *recvmsg()*, *sendmsg()*, *setitimer()*, *ualarm()*,
 30999 *write()*, the Base Definitions volume of IEEE Std 1003.1-2001, `<sys/select.h>`, `<sys/time.h>`

31000 CHANGE HISTORY

31001 First released in Issue 4, Version 2.

31002 Issue 5

31003 Moved from X/OPEN UNIX extension to BASE.

31004 In the ERRORS section, the text has been changed to indicate that [EINVAL] is returned when
 31005 *nfds* is less than 0 or greater than FD_SETSIZE. It previously stated less than 0, or greater than or
 31006 equal to FD_SETSIZE.

31007 Text about *timeout* is moved from the APPLICATION USAGE section to the DESCRIPTION.

31008 Issue 6

31009 The Open Group Corrigendum U026/6 is applied, changing the occurrences of *readfs* and *writfs*
 31010 in the *select()* DESCRIPTION to be *readfds* and *writefds*.

31011 Text referring to sockets is added to the DESCRIPTION.

31012 The DESCRIPTION and ERRORS sections are updated so that references to STREAMS are
 31013 marked as part of the XSI STREAMS Option Group.

31014 The following new requirements on POSIX implementations derive from alignment with the
 31015 Single UNIX Specification:

- 31016 • These functions are now mandatory.

31017 The *pselect()* function is added for alignment with IEEE Std 1003.1g-2000 and additional detail
 31018 related to sockets semantics is added to the DESCRIPTION.

31019 The *select()* function now requires inclusion of `<sys/select.h>`.
31020 The **restrict** keyword is added to the *select()* prototype for alignment with the
31021 ISO/IEC 9899:1999 standard.

31022 **NAME**

31023 pthread_atfork — register fork handlers

31024 **SYNOPSIS**

31025 THR #include <pthread.h>

```
31026 int pthread_atfork(void (*prepare)(void), void (*parent)(void),
31027 void (*child)(void));
```

31028

31029 **DESCRIPTION**

31030 The *pthread_atfork()* function shall declare fork handlers to be called before and after *fork()*, in
 31031 the context of the thread that called *fork()*. The *prepare* fork handler shall be called before *fork()*
 31032 processing commences. The *parent* fork handler shall be called after *fork()* processing completes
 31033 in the parent process. The *child* fork handler shall be called after *fork()* processing completes in
 31034 the child process. If no handling is desired at one or more of these three points, the
 31035 corresponding fork handler address(es) may be set to NULL.

31036 The order of calls to *pthread_atfork()* is significant. The *parent* and *child* fork handlers shall be
 31037 called in the order in which they were established by calls to *pthread_atfork()*. The *prepare* fork
 31038 handlers shall be called in the opposite order.

31039 **RETURN VALUE**

31040 Upon successful completion, *pthread_atfork()* shall return a value of zero; otherwise, an error
 31041 number shall be returned to indicate the error.

31042 **ERRORS**31043 The *pthread_atfork()* function shall fail if:

31044 [ENOMEM] Insufficient table space exists to record the fork handler addresses.

31045 The *pthread_atfork()* function shall not return an error code of [EINTR].31046 **EXAMPLES**

31047 None.

31048 **APPLICATION USAGE**

31049 None.

31050 **RATIONALE**

31051 There are at least two serious problems with the semantics of *fork()* in a multi-threaded
 31052 program. One problem has to do with state (for example, memory) covered by mutexes.
 31053 Consider the case where one thread has a mutex locked and the state covered by that mutex is
 31054 inconsistent while another thread calls *fork()*. In the child, the mutex is in the locked state
 31055 (locked by a nonexistent thread and thus can never be unlocked). Having the child simply
 31056 reinitialize the mutex is unsatisfactory since this approach does not resolve the question about
 31057 how to correct or otherwise deal with the inconsistent state in the child.

31058 It is suggested that programs that use *fork()* call an *exec* function very soon afterwards in the
 31059 child process, thus resetting all states. In the meantime, only a short list of async-signal-safe
 31060 library routines are promised to be available.

31061 Unfortunately, this solution does not address the needs of multi-threaded libraries. Application
 31062 programs may not be aware that a multi-threaded library is in use, and they feel free to call any
 31063 number of library routines between the *fork()* and *exec* calls, just as they always have. Indeed,
 31064 they may be extant single-threaded programs and cannot, therefore, be expected to obey new
 31065 restrictions imposed by the threads library.

On the other hand, the multi-threaded library needs a way to protect its internal state during *fork()* in case it is re-entered later in the child process. The problem arises especially in multi-threaded I/O libraries, which are almost sure to be invoked between the *fork()* and *exec* calls to effect I/O redirection. The solution may require locking mutex variables during *fork()*, or it may entail simply resetting the state in the child after the *fork()* processing completes.

The *pthread_atfork()* function provides multi-threaded libraries with a means to protect themselves from innocent application programs that call *fork()*, and it provides multi-threaded application programs with a standard mechanism for protecting themselves from *fork()* calls in a library routine or the application itself.

The expected usage is that the *prepare* handler acquires all mutex locks and the other two fork handlers release them.

For example, an application can supply a *prepare* routine that acquires the necessary mutexes the library maintains and supply *child* and *parent* routines that release those mutexes, thus ensuring that the child gets a consistent snapshot of the state of the library (and that no mutexes are left stranded). Alternatively, some libraries might be able to supply just a *child* routine that reinitializes the mutexes in the library and all associated states to some known value (for example, what it was when the image was originally executed).

When *fork()* is called, only the calling thread is duplicated in the child process. Synchronization variables remain in the same state in the child as they were in the parent at the time *fork()* was called. Thus, for example, mutex locks may be held by threads that no longer exist in the child process, and any associated states may be inconsistent. The parent process may avoid this by explicit code that acquires and releases locks critical to the child via *pthread_atfork()*. In addition, any critical threads need to be recreated and reinitialized to the proper state in the child (also via *pthread_atfork()*).

A higher-level package may acquire locks on its own data structures before invoking lower-level packages. Under this scenario, the order specified for fork handler calls allows a simple rule of initialization for avoiding package deadlock: a package initializes all packages on which it depends before it calls the *pthread_atfork()* function for itself.

31094 FUTURE DIRECTIONS

31095 None.

31096 SEE ALSO

31097 *atexit()*, *fork()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>

31098 CHANGE HISTORY

31099 First released in Issue 5. Derived from the POSIX Threads Extension.

31100 IEEE PASC Interpretation 1003.1c #4 is applied.

31101 Issue 6

31102 The *pthread_atfork()* function is marked as part of the Threads option.

31103 The <pthread.h> header is added to the SYNOPSIS.

31104 **NAME**

31105 pthread_attr_destroy, pthread_attr_init — destroy and initialize the thread attributes object

31106 **SYNOPSIS**31107 THR `#include <pthread.h>`31108 `int pthread_attr_destroy(pthread_attr_t *attr);`31109 `int pthread_attr_init(pthread_attr_t *attr);`

31110

31111 **DESCRIPTION**

31112 The *pthread_attr_destroy()* function shall destroy a thread attributes object. An implementation
 31113 may cause *pthread_attr_destroy()* to set *attr* to an implementation-defined invalid value. A
 31114 destroyed *attr* attributes object can be reinitialized using *pthread_attr_init()*; the results of
 31115 otherwise referencing the object after it has been destroyed are undefined.

31116 The *pthread_attr_init()* function shall initialize a thread attributes object *attr* with the default
 31117 value for all of the individual attributes used by a given implementation.

31118 The resulting attributes object (possibly modified by setting individual attribute values) when
 31119 used by *pthread_create()* defines the attributes of the thread created. A single attributes object can
 31120 be used in multiple simultaneous calls to *pthread_create()*. Results are undefined if
 31121 *pthread_attr_init()* is called specifying an already initialized *attr* attributes object.

31122 **RETURN VALUE**

31123 Upon successful completion, *pthread_attr_destroy()* and *pthread_attr_init()* shall return a value of
 31124 0; otherwise, an error number shall be returned to indicate the error.

31125 **ERRORS**31126 The *pthread_attr_init()* function shall fail if:

31127 [ENOMEM] Insufficient memory exists to initialize the thread attributes object.

31128 These functions shall not return an error code of [EINTR].

31129 **EXAMPLES**

31130 None.

31131 **APPLICATION USAGE**

31132 None.

31133 **RATIONALE**

31134 Attributes objects are provided for threads, mutexes, and condition variables as a mechanism to
 31135 support probable future standardization in these areas without requiring that the function itself
 31136 be changed.

31137 Attributes objects provide clean isolation of the configurable aspects of threads. For example,
 31138 “stack size” is an important attribute of a thread, but it cannot be expressed portably. When
 31139 porting a threaded program, stack sizes often need to be adjusted. The use of attributes objects
 31140 can help by allowing the changes to be isolated in a single place, rather than being spread across
 31141 every instance of thread creation.

31142 Attributes objects can be used to set up “classes” of threads with similar attributes; for example,
 31143 “threads with large stacks and high priority” or “threads with minimal stacks”. These classes
 31144 can be defined in a single place and then referenced wherever threads need to be created.
 31145 Changes to “class” decisions become straightforward, and detailed analysis of each
 31146 *pthread_create()* call is not required.

31147 The attributes objects are defined as opaque types as an aid to extensibility. If these objects had
 31148 been specified as structures, adding new attributes would force recompilation of all multi-

31149 threaded programs when the attributes objects are extended; this might not be possible if
 31150 different program components were supplied by different vendors.

31151 Additionally, opaque attributes objects present opportunities for improving performance.
 31152 Argument validity can be checked once when attributes are set, rather than each time a thread is
 31153 created. Implementations often need to cache kernel objects that are expensive to create.
 31154 Opaque attributes objects provide an efficient mechanism to detect when cached objects become
 31155 invalid due to attribute changes.

31156 Since assignment is not necessarily defined on a given opaque type, implementation-defined
 31157 default values cannot be defined in a portable way. The solution to this problem is to allow
 31158 attributes objects to be initialized dynamically by attributes object initialization functions, so
 31159 that default values can be supplied automatically by the implementation.

31160 The following proposal was provided as a suggested alternative to the supplied attributes:

- 31161 1. Maintain the style of passing a parameter formed by the bitwise-inclusive OR of flags to
 31162 the initialization routines (*pthread_create()*, *pthread_mutex_init()*, *pthread_cond_init()*). The
 31163 parameter containing the flags should be an opaque type for extensibility. If no flags are
 31164 set in the parameter, then the objects are created with default characteristics. An
 31165 implementation may specify implementation-defined flag values and associated behavior.
- 31166 2. If further specialization of mutexes and condition variables is necessary, implementations
 31167 may specify additional procedures that operate on the **pthread_mutex_t** and
 31168 **pthread_cond_t** objects (instead of on attributes objects).

31169 The difficulties with this solution are:

- 31170 1. A bitmask is not opaque if bits have to be set into bitvector attributes objects using
 31171 explicitly-coded bitwise-inclusive OR operations. If the set of options exceeds an **int**,
 31172 application programmers need to know the location of each bit. If bits are set or read by
 31173 encapsulation (that is, get and set functions), then the bitmask is merely an
 31174 implementation of attributes objects as currently defined and should not be exposed to the
 31175 programmer.
- 31176 2. Many attributes are not Boolean or very small integral values. For example, scheduling
 31177 policy may be placed in 3-bit or 4-bit, but priority requires 5-bit or more, thereby taking up
 31178 at least 8 bits out of a possible 16 bits on machines with 16-bit integers. Because of this, the
 31179 bitmask can only reasonably control whether particular attributes are set or not, and it
 31180 cannot serve as the repository of the value itself. The value needs to be specified as a
 31181 function parameter (which is non-extensible), or by setting a structure field (which is non-
 31182 opaque), or by get and set functions (making the bitmask a redundant addition to the
 31183 attributes objects).

31184 Stack size is defined as an optional attribute because the very notion of a stack is inherently
 31185 machine-dependent. Some implementations may not be able to change the size of the stack, for
 31186 example, and others may not need to because stack pages may be discontinuous and can be
 31187 allocated and released on demand.

31188 The attribute mechanism has been designed in large measure for extensibility. Future extensions
 31189 to the attribute mechanism or to any attributes object defined in this volume of
 31190 IEEE Std 1003.1-2001 has to be done with care so as not to affect binary-compatibility.

31191 Attributes objects, even if allocated by means of dynamic allocation functions such as *malloc()*,
 31192 may have their size fixed at compile time. This means, for example, a *pthread_create()* in an
 31193 implementation with extensions to **pthread_attr_t** cannot look beyond the area that the binary
 31194 application assumes is valid. This suggests that implementations should maintain a size field in
 31195 the attributes object, as well as possibly version information, if extensions in different directions

31196 (possibly by different vendors) are to be accommodated.

31197 **FUTURE DIRECTIONS**

31198 None.

31199 **SEE ALSO**

31200 *pthread_attr_getstackaddr()*, *pthread_attr_getstacksize()*, *pthread_attr_getdetachstate()*,
31201 *pthread_create()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>

31202 **CHANGE HISTORY**

31203 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

31204 **Issue 6**

31205 The *pthread_attr_destroy()* and *pthread_attr_init()* functions are marked as part of the Threads
31206 option.

31207 IEEE PASC Interpretation 1003.1 #107 is applied, noting that the effect of initializing an already
31208 initialized thread attributes object is undefined.

31209 **NAME**

31210 pthread_attr_getdetachstate, pthread_attr_setdetachstate — get and set the detachstate attribute

31211 **SYNOPSIS**

```
31212 THR      #include <pthread.h>

31213          int pthread_attr_getdetachstate(const pthread_attr_t *attr,
31214          int *detachstate);
31215          int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
31216
```

31217 **DESCRIPTION**

31218 The *detachstate* attribute controls whether the thread is created in a detached state. If the thread
 31219 is created detached, then use of the ID of the newly created thread by the *pthread_detach()* or
 31220 *pthread_join()* function is an error.

31221 The *pthread_attr_getdetachstate()* and *pthread_attr_setdetachstate()* functions, respectively, shall
 31222 get and set the *detachstate* attribute in the *attr* object.

31223 For *pthread_attr_getdetachstate()*, *detachstate* shall be set to either
 31224 PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE.

31225 For *pthread_attr_setdetachstate()*, the application shall set *detachstate* to either
 31226 PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE.

31227 A value of PTHREAD_CREATE_DETACHED shall cause all threads created with *attr* to be in
 31228 the detached state, whereas using a value of PTHREAD_CREATE_JOINABLE shall cause all
 31229 threads created with *attr* to be in the joinable state. The default value of the *detachstate* attribute
 31230 shall be PTHREAD_CREATE_JOINABLE.

31231 **RETURN VALUE**

31232 Upon successful completion, *pthread_attr_getdetachstate()* and *pthread_attr_setdetachstate()* shall
 31233 return a value of 0; otherwise, an error number shall be returned to indicate the error.

31234 The *pthread_attr_getdetachstate()* function stores the value of the *detachstate* attribute in *detachstate*
 31235 if successful.

31236 **ERRORS**

31237 The *pthread_attr_setdetachstate()* function shall fail if:

31238 [EINVAL] The value of *detachstate* was not valid

31239 These functions shall not return an error code of [EINTR].

31240 **EXAMPLES**

31241 None.

31242 **APPLICATION USAGE**

31243 None.

31244 **RATIONALE**

31245 None.

31246 **FUTURE DIRECTIONS**

31247 None.

31248 **SEE ALSO**

31249 *pthread_attr_destroy()*, *pthread_attr_getstackaddr()*, *pthread_attr_getstacksize()*, *pthread_create()*, the
 31250 Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

31251 **CHANGE HISTORY**

31252 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

31253 **Issue 6**

31254 The *pthread_attr_setdetachstate()* and *pthread_attr_getdetachstate()* functions are marked as part of
31255 the Threads option.

31256 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

31257 **NAME**

31258 pthread_attr_getguardsize, pthread_attr_setguardsize — get and set the thread guardsize
 31259 attribute

31260 **SYNOPSIS**

31261 XSI #include <pthread.h>

```
31262 int pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
31263                             size_t *restrict guardsize);
31264 int pthread_attr_setguardsize(pthread_attr_t *attr,
31265                             size_t guardsize);
31266
```

31267 **DESCRIPTION**

31268 The *pthread_attr_getguardsize()* function shall get the *guardsize* attribute in the *attr* object. This
 31269 attribute shall be returned in the *guardsize* parameter.

31270 The *pthread_attr_setguardsize()* function shall set the *guardsize* attribute in the *attr* object. The new
 31271 value of this attribute shall be obtained from the *guardsize* parameter. If *guardsize* is zero, a guard
 31272 area shall not be provided for threads created with *attr*. If *guardsize* is greater than zero, a guard
 31273 area of at least size *guardsize* bytes shall be provided for each thread created with *attr*.

31274 The *guardsize* attribute controls the size of the guard area for the created thread's stack. The
 31275 *guardsize* attribute provides protection against overflow of the stack pointer. If a thread's stack is
 31276 created with guard protection, the implementation allocates extra memory at the overflow end
 31277 of the stack as a buffer against stack overflow of the stack pointer. If an application overflows
 31278 into this buffer an error shall result (possibly in a SIGSEGV signal being delivered to the thread).

31279 A conforming implementation may round up the value contained in *guardsize* to a multiple of
 31280 the configurable system variable {PAGESIZE} (see <sys/mman.h>). If an implementation
 31281 rounds up the value of *guardsize* to a multiple of {PAGESIZE}, a call to *pthread_attr_getguardsize()*
 31282 specifying *attr* shall store in the *guardsize* parameter the guard size specified by the previous
 31283 *pthread_attr_setguardsize()* function call.

31284 The default value of the *guardsize* attribute is {PAGESIZE} bytes. The actual value of {PAGESIZE}
 31285 is implementation-defined.

31286 If the *stackaddr* or *stack* attribute has been set (that is, the caller is allocating and managing its
 31287 own thread stacks), the *guardsize* attribute shall be ignored and no protection shall be provided
 31288 by the implementation. It is the responsibility of the application to manage stack overflow along
 31289 with stack allocation and management in this case.

31290 **RETURN VALUE**

31291 If successful, the *pthread_attr_getguardsize()* and *pthread_attr_setguardsize()* functions shall return
 31292 zero; otherwise, an error number shall be returned to indicate the error.

31293 **ERRORS**

31294 The *pthread_attr_getguardsize()* and *pthread_attr_setguardsize()* functions shall fail if:

31295 [EINVAL] The attribute *attr* is invalid.

31296 [EINVAL] The parameter *guardsize* is invalid.

31297 These functions shall not return an error code of [EINTR].

31298 **EXAMPLES**

31299 None.

31300 **APPLICATION USAGE**

31301 None.

31302 **RATIONALE**31303 The *guardsize* attribute is provided to the application for two reasons:

- 31304 1. Overflow protection can potentially result in wasted system resources. An application
31305 that creates a large number of threads, and which knows its threads never overflow their
31306 stack, can save system resources by turning off guard areas.
- 31307 2. When threads allocate large data structures on the stack, large guard areas may be needed
31308 to detect stack overflow.

31309 **FUTURE DIRECTIONS**

31310 None.

31311 **SEE ALSO**

31312 The Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>, <sys/mman.h>

31313 **CHANGE HISTORY**

31314 First released in Issue 5.

31315 **Issue 6**31316 In the ERRORS section, a third [EINVAL] error condition is removed as it is covered by the
31317 second error condition.31318 The **restrict** keyword is added to the *pthread_attr_getguardsize()* prototype for alignment with the
31319 ISO/IEC 9899:1999 standard.

31320 **NAME**

31321 pthread_attr_getinheritsched, pthread_attr_setinheritsched — get and set the inheritsched
 31322 attribute (**REALTIME THREADS**)

31323 **SYNOPSIS**

31324 THR TPS #include <pthread.h>

```
31325 int pthread_attr_getinheritsched(const pthread_attr_t *restrict attr,
31326     int *restrict inheritsched);
31327 int pthread_attr_setinheritsched(pthread_attr_t *attr,
31328     int inheritsched);
31329
```

31330 **DESCRIPTION**

31331 The *pthread_attr_getinheritsched()*, and *pthread_attr_setinheritsched()* functions, respectively, shall
 31332 get and set the *inheritsched* attribute in the *attr* argument.

31333 When the attributes objects are used by *pthread_create()*, the *inheritsched* attribute determines
 31334 how the other scheduling attributes of the created thread shall be set.

31335 **PTHREAD_INHERIT_SCHED**

31336 Specifies that the thread scheduling attributes shall be inherited from the creating thread,
 31337 and the scheduling attributes in this *attr* argument shall be ignored.

31338 **PTHREAD_EXPLICIT_SCHED**

31339 Specifies that the thread scheduling attributes shall be set to the corresponding values from
 31340 this attributes object.

31341 The symbols PTHREAD_INHERIT_SCHED and PTHREAD_EXPLICIT_SCHED are defined in
 31342 the <pthread.h> header.

31343 The following thread scheduling attributes defined by IEEE Std 1003.1-2001 are affected by the
 31344 *inheritsched* attribute: scheduling policy (*schedpolicy*), scheduling parameters (*schedparam*), and
 31345 scheduling contention scope (*contentionscope*).

31346 **RETURN VALUE**

31347 If successful, the *pthread_attr_getinheritsched()* and *pthread_attr_setinheritsched()* functions shall
 31348 return zero; otherwise, an error number shall be returned to indicate the error.

31349 **ERRORS**

31350 The *pthread_attr_setinheritsched()* function may fail if:

31351 [EINVAL] The value of *inheritsched* is not valid.

31352 [ENOTSUP] An attempt was made to set the attribute to an unsupported value.

31353 These functions shall not return an error code of [EINTR].

31354 **EXAMPLES**

31355 None.

31356 **APPLICATION USAGE**

31357 After these attributes have been set, a thread can be created with the specified attributes using
 31358 *pthread_create()*. Using these routines does not affect the current running thread.

31359 **RATIONALE**

31360 None.

31361 **FUTURE DIRECTIONS**

31362 None.

31363 **SEE ALSO**

31364 *pthread_attr_destroy()*, *pthread_attr_getscope()*, *pthread_attr_getschedpolicy()*,
31365 *pthread_attr_getschedparam()*, *pthread_create()*, the Base Definitions volume of
31366 IEEE Std 1003.1-2001, <pthread.h>, <sched.h>

31367 **CHANGE HISTORY**

31368 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

31369 Marked as part of the Realtime Threads Feature Group.

31370 **Issue 6**

31371 The *pthread_attr_getinheritsched()* and *pthread_attr_setinheritsched()* functions are marked as part
31372 of the Threads and Thread Execution Scheduling options.

31373 The [ENOSYS] error condition has been removed as stubs need not be provided if an
31374 implementation does not support the Thread Execution Scheduling option.

31375 The **restrict** keyword is added to the *pthread_attr_getinheritsched()* prototype for alignment with
31376 the ISO/IEC 9899:1999 standard.

31377 **NAME**

31378 pthread_attr_getschedparam, pthread_attr_setschedparam — get and set the schedparam
 31379 attribute

31380 **SYNOPSIS**

31381 THR #include <pthread.h>

```
31382 int pthread_attr_getschedparam(const pthread_attr_t *restrict attr,  
31383 struct sched_param *restrict param);  
31384 int pthread_attr_setschedparam(pthread_attr_t *restrict attr,  
31385 const struct sched_param *restrict param);  
31386
```

31387 **DESCRIPTION**

31388 The *pthread_attr_getschedparam()*, and *pthread_attr_setschedparam()* functions, respectively, shall
 31389 get and set the scheduling parameter attributes in the *attr* argument. The contents of the *param*
 31390 structure are defined in the <sched.h> header. For the SCHED_FIFO and SCHED_RR policies,
 31391 the only required member of *param* is *sched_priority*.

31392 TSP For the SCHED_SPORADIC policy, the required members of the *param* structure are
 31393 *sched_priority*, *sched_ss_low_priority*, *sched_ss_repl_period*, *sched_ss_init_budget*, and
 31394 *sched_ss_max_repl*. The specified *sched_ss_repl_period* must be greater than or equal to the
 31395 specified *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail.
 31396 The value of *sched_ss_max_repl* shall be within the inclusive range [1,{SS_REPL_MAX}] for the
 31397 function to succeed; if not, the function shall fail.

31398 **RETURN VALUE**

31399 If successful, the *pthread_attr_getschedparam()* and *pthread_attr_setschedparam()* functions shall
 31400 return zero; otherwise, an error number shall be returned to indicate the error.

31401 **ERRORS**

31402 The *pthread_attr_setschedparam()* function may fail if:

31403 [EINVAL] The value of *param* is not valid.

31404 [ENOTSUP] An attempt was made to set the attribute to an unsupported value.

31405 These functions shall not return an error code of [EINTR].

31406 **EXAMPLES**

31407 None.

31408 **APPLICATION USAGE**

31409 After these attributes have been set, a thread can be created with the specified attributes using
 31410 *pthread_create()*. Using these routines does not affect the current running thread.

31411 **RATIONALE**

31412 None.

31413 **FUTURE DIRECTIONS**

31414 None.

31415 **SEE ALSO**

31416 *pthread_attr_destroy()*, *pthread_attr_getscope()*, *pthread_attr_getinheritsched()*,
 31417 *pthread_attr_getschedpolicy()*, *pthread_create()*, the Base Definitions volume of
 31418 IEEE Std 1003.1-2001, <pthread.h>, <sched.h>

31419 CHANGE HISTORY

31420 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

31421 Issue 6

31422 The *pthread_attr_getschedparam()* and *pthread_attr_setschedparam()* functions are marked as part
31423 of the Threads option.

31424 The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

31425 The **restrict** keyword is added to the *pthread_attr_getschedparam()* and
31426 *pthread_attr_setschedparam()* prototypes for alignment with the ISO/IEC 9899:1999 standard.

31427 **NAME**

31428 pthread_attr_getschedpolicy, pthread_attr_setschedpolicy — get and set the schedpolicy
 31429 attribute (**REALTIME THREADS**)

31430 **SYNOPSIS**

31431 THR TPS #include <pthread.h>

```
31432 int pthread_attr_getschedpolicy(const pthread_attr_t *restrict attr,
31433 int *restrict policy);
31434 int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
31435
```

31436 **DESCRIPTION**

31437 The *pthread_attr_getschedpolicy()* and *pthread_attr_setschedpolicy()* functions, respectively, shall
 31438 get and set the *schedpolicy* attribute in the *attr* argument.

31439 The supported values of *policy* shall include SCHED_FIFO, SCHED_RR, and SCHED_OTHER,
 31440 which are defined in the <sched.h> header. When threads executing with the scheduling policy
 31441 TSP SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC are waiting on a mutex, they shall acquire
 31442 the mutex in priority order when the mutex is unlocked.

31443 **RETURN VALUE**

31444 If successful, the *pthread_attr_getschedpolicy()* and *pthread_attr_setschedpolicy()* functions shall
 31445 return zero; otherwise, an error number shall be returned to indicate the error.

31446 **ERRORS**

31447 The *pthread_attr_setschedpolicy()* function may fail if:

31448 [EINVAL] The value of *policy* is not valid.

31449 [ENOTSUP] An attempt was made to set the attribute to an unsupported value.

31450 These functions shall not return an error code of [EINTR].

31451 **EXAMPLES**

31452 None.

31453 **APPLICATION USAGE**

31454 After these attributes have been set, a thread can be created with the specified attributes using
 31455 *pthread_create()*. Using these routines does not affect the current running thread.

31456 **RATIONALE**

31457 None.

31458 **FUTURE DIRECTIONS**

31459 None.

31460 **SEE ALSO**

31461 *pthread_attr_destroy()*, *pthread_attr_getscope()*, *pthread_attr_getinheritsched()*,
 31462 *pthread_attr_getschedparam()*, *pthread_create()*, the Base Definitions volume of
 31463 IEEE Std 1003.1-2001, <pthread.h>, <sched.h>

31464 **CHANGE HISTORY**

31465 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

31466 Marked as part of the Realtime Threads Feature Group.

31467 Issue 6

31468 The *pthread_attr_getschedpolicy()* and *pthread_attr_setschedpolicy()* functions are marked as part of
31469 the Threads and Thread Execution Scheduling options.

31470 The [ENOSYS] error condition has been removed as stubs need not be provided if an
31471 implementation does not support the Thread Execution Scheduling option.

31472 The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

31473 The **restrict** keyword is added to the *pthread_attr_getschedpolicy()* prototype for alignment with
31474 the ISO/IEC 9899:1999 standard.

31475 **NAME**

31476 pthread_attr_getscope, pthread_attr_setscope — get and set the contentionscope attribute
 31477 (**REALTIME THREADS**)

31478 **SYNOPSIS**

31479 THR TPS #include <pthread.h>

```
31480 int pthread_attr_getscope(const pthread_attr_t *restrict attr,
31481 int *restrict contentionscope);
31482 int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
31483
```

31484 **DESCRIPTION**

31485 The *pthread_attr_getscope()* and *pthread_attr_setscope()* functions, respectively, shall get and set
 31486 the *contentionscope* attribute in the *attr* object.

31487 The *contentionscope* attribute may have the values PTHREAD_SCOPE_SYSTEM, signifying
 31488 system scheduling contention scope, or PTHREAD_SCOPE_PROCESS, signifying process
 31489 scheduling contention scope. The symbols PTHREAD_SCOPE_SYSTEM and
 31490 PTHREAD_SCOPE_PROCESS are defined in the <pthread.h> header.

31491 **RETURN VALUE**

31492 If successful, the *pthread_attr_getscope()* and *pthread_attr_setscope()* functions shall return zero;
 31493 otherwise, an error number shall be returned to indicate the error.

31494 **ERRORS**

31495 The *pthread_attr_setscope()* function may fail if:

- 31496 [EINVAL] The value of *contentionscope* is not valid.
- 31497 [ENOTSUP] An attempt was made to set the attribute to an unsupported value.
- 31498 These functions shall not return an error code of [EINTR].

31499 **EXAMPLES**

31500 None.

31501 **APPLICATION USAGE**

31502 After these attributes have been set, a thread can be created with the specified attributes using
 31503 *pthread_create()*. Using these routines does not affect the current running thread.

31504 **RATIONALE**

31505 None.

31506 **FUTURE DIRECTIONS**

31507 None.

31508 **SEE ALSO**

31509 *pthread_attr_destroy()*, *pthread_attr_getinheritsched()*, *pthread_attr_getschedpolicy()*,
 31510 *pthread_attr_getschedparam()*, *pthread_create()*, the Base Definitions volume of
 31511 IEEE Std 1003.1-2001, <pthread.h>, <sched.h>

31512 **CHANGE HISTORY**

- 31513 First released in Issue 5. Included for alignment with the POSIX Threads Extension.
- 31514 Marked as part of the Realtime Threads Feature Group.

Issue 6

The *pthread_attr_getscope()* and *pthread_attr_setscope()* functions are marked as part of the Threads and Thread Execution Scheduling options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Execution Scheduling option.

The **restrict** keyword is added to the *pthread_attr_getscope()* prototype for alignment with the ISO/IEC 9899:1999 standard.

31522 **NAME**

31523 pthread_attr_getstack, pthread_attr_setstack — get and set stack attributes

31524 **SYNOPSIS**

31525 THR #include <pthread.h>

```

31526 TSA TSS int pthread_attr_getstack(const pthread_attr_t *restrict attr,
31527 void **restrict stackaddr, size_t *restrict stacksize);
31528 int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr,
31529 size_t stacksize);
31530

```

31531 **DESCRIPTION**

31532 The *pthread_attr_getstack()* and *pthread_attr_setstack()* functions, respectively, shall get and set
 31533 the thread creation stack attributes *stackaddr* and *stacksize* in the *attr* object.

31534 The stack attributes specify the area of storage to be used for the created thread's stack. The base
 31535 (lowest addressable byte) of the storage shall be *stackaddr*, and the size of the storage shall be
 31536 *stacksize* bytes. The *stacksize* shall be at least {PTHREAD_STACK_MIN}. The *stackaddr* shall be
 31537 aligned appropriately to be used as a stack; for example, *pthread_attr_setstack()* may fail with
 31538 [EINVAL] if (*stackaddr* & 0x7) is not 0. All pages within the stack described by *stackaddr* and
 31539 *stacksize* shall be both readable and writable by the thread.

31540 **RETURN VALUE**

31541 Upon successful completion, these functions shall return a value of 0; otherwise, an error
 31542 number shall be returned to indicate the error.

31543 The *pthread_attr_getstack()* function shall store the stack attribute values in *stackaddr* and *stacksize*
 31544 if successful.

31545 **ERRORS**

31546 The *pthread_attr_setstack()* function shall fail if:

31547 [EINVAL] The value of *stacksize* is less than {PTHREAD_STACK_MIN} or exceeds an
 31548 implementation-defined limit.

31549 The *pthread_attr_setstack()* function may fail if:

31550 [EINVAL] The value of *stackaddr* does not have proper alignment to be used as a stack, or
 31551 if (*stackaddr* + *stacksize*) lacks proper alignment.

31552 [EACCES] The stack page(s) described by *stackaddr* and *stacksize* are not both readable
 31553 and writable by the thread.

31554 These functions shall not return an error code of [EINTR].

31555 **EXAMPLES**

31556 None.

31557 **APPLICATION USAGE**

31558 These functions are appropriate for use by applications in an environment where the stack for a
 31559 thread must be placed in some particular region of memory.

31560 While it might seem that an application could detect stack overflow by providing a protected
 31561 page outside the specified stack region, this cannot be done portably. Implementations are free
 31562 to place the thread's initial stack pointer anywhere within the specified region to accommodate
 31563 the machine's stack pointer behavior and allocation requirements. Furthermore, on some
 31564 architectures, such as the IA-64, "overflow" might mean that two separate stack pointers
 31565 allocated within the region will overlap somewhere in the middle of the region.

31566 **RATIONALE**

31567 None.

31568 **FUTURE DIRECTIONS**

31569 None.

31570 **SEE ALSO**31571 *pthread_attr_init()*, *pthread_attr_setdetachstate()*, *pthread_attr_setstacksize()*, *pthread_create()*, the
31572 Base Definitions volume of IEEE Std 1003.1-2001, <limits.h>, <pthread.h>31573 **CHANGE HISTORY**31574 First released in Issue 6. Developed as an XSI extension and brought into the BASE by IEEE
31575 PASC Interpretation 1003.1 #101.

31576 **NAME**

31577 pthread_attr_getstackaddr, pthread_attr_setstackaddr — get and set the stackaddr attribute

31578 **SYNOPSIS**

31579 THR TSA #include <pthread.h>

```

31580 OB      int pthread_attr_getstackaddr(const pthread_attr_t *restrict attr,
31581          void **restrict stackaddr);
31582          int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
31583

```

31584 **DESCRIPTION**

31585 The *pthread_attr_getstackaddr()* and *pthread_attr_setstackaddr()* functions, respectively, shall get
 31586 and set the thread creation *stackaddr* attribute in the *attr* object.

31587 The *stackaddr* attribute specifies the location of storage to be used for the created thread's stack.
 31588 The size of the storage shall be at least {PTHREAD_STACK_MIN}.

31589 **RETURN VALUE**

31590 Upon successful completion, *pthread_attr_getstackaddr()* and *pthread_attr_setstackaddr()* shall
 31591 return a value of 0; otherwise, an error number shall be returned to indicate the error.

31592 The *pthread_attr_getstackaddr()* function stores the *stackaddr* attribute value in *stackaddr* if
 31593 successful.

31594 **ERRORS**

31595 No errors are defined.

31596 These functions shall not return an error code of [EINTR].

31597 **EXAMPLES**

31598 None.

31599 **APPLICATION USAGE**

31600 The specification of the *stackaddr* attribute presents several ambiguities that make portable use of
 31601 these interfaces impossible. The description of the single address parameter as a “stack” does
 31602 not specify a particular relationship between the address and the “stack” implied by that
 31603 address. For example, the address may be taken as the low memory address of a buffer intended
 31604 for use as a stack, or it may be taken as the address to be used as the initial stack pointer register
 31605 value for the new thread. These two are not the same except for a machine on which the stack
 31606 grows “up” from low memory to high, and on which a “push” operation first stores the value in
 31607 memory and then increments the stack pointer register. Further, on a machine where the stack
 31608 grows “down” from high memory to low, interpretation of the address as the “low memory”
 31609 address requires a determination of the intended size of the stack. IEEE Std 1003.1-2001 has
 31610 introduced the new interfaces *pthread_attr_setstack()* and *pthread_attr_getstack()* to resolve these
 31611 ambiguities.

31612 **RATIONALE**

31613 None.

31614 **FUTURE DIRECTIONS**

31615 None.

31616 **SEE ALSO**

31617 *pthread_attr_destroy()*, *pthread_attr_getdetachstate()*, *pthread_attr_getstack()*,
 31618 *pthread_attr_getstacksize()*, *pthread_attr_setstack()*, *pthread_create()*, the Base Definitions volume
 31619 of IEEE Std 1003.1-2001, <limits.h>, <pthread.h>

31620 CHANGE HISTORY

31621 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

31622 Issue 6

31623 The *pthread_attr_getstackaddr()* and *pthread_attr_setstackaddr()* functions are marked as part of
31624 the Threads and Thread Stack Address Attribute options.

31625 The **restrict** keyword is added to the *pthread_attr_getstackaddr()* prototype for alignment with the
31626 ISO/IEC 9899:1999 standard.

31627 These functions are marked obsolescent.

31628 **NAME**

31629 pthread_attr_getstacksize, pthread_attr_setstacksize — get and set the stacksize attribute

31630 **SYNOPSIS**

31631 THR TSA #include <pthread.h>

```

31632 int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
31633     size_t *restrict stacksize);
31634 int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
31635

```

31636 **DESCRIPTION**31637 The *pthread_attr_getstacksize()* and *pthread_attr_setstacksize()* functions, respectively, shall get
31638 and set the thread creation *stacksize* attribute in the *attr* object.31639 The *stacksize* attribute shall define the minimum stack size (in bytes) allocated for the created
31640 threads stack.31641 **RETURN VALUE**31642 Upon successful completion, *pthread_attr_getstacksize()* and *pthread_attr_setstacksize()* shall
31643 return a value of 0; otherwise, an error number shall be returned to indicate the error.31644 The *pthread_attr_getstacksize()* function stores the *stacksize* attribute value in *stacksize* if
31645 successful.31646 **ERRORS**31647 The *pthread_attr_setstacksize()* function shall fail if:31648 [EINVAL] The value of *stacksize* is less than {PTHREAD_STACK_MIN} or exceeds a
31649 system-imposed limit.

31650 These functions shall not return an error code of [EINTR].

31651 **EXAMPLES**

31652 None.

31653 **APPLICATION USAGE**

31654 None.

31655 **RATIONALE**

31656 None.

31657 **FUTURE DIRECTIONS**

31658 None.

31659 **SEE ALSO**31660 *pthread_attr_destroy()*, *pthread_attr_getstackaddr()*, *pthread_attr_getdetachstate()*, *pthread_create()*,
31661 the Base Definitions volume of IEEE Std 1003.1-2001, <limits.h>, <pthread.h>31662 **CHANGE HISTORY**

31663 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

31664 **Issue 6**31665 The *pthread_attr_getstacksize()* and *pthread_attr_setstacksize()* functions are marked as part of the
31666 Threads and Thread Stack Address Attribute options.31667 The **restrict** keyword is added to the *pthread_attr_getstacksize()* prototype for alignment with the
31668 ISO/IEC 9899:1999 standard.

31669 NAME

31670 pthread_attr_init — initialize the thread attributes object

31671 SYNOPSIS

31672 THR `#include <pthread.h>`

31673 `int pthread_attr_init(pthread_attr_t *attr);`

31674

31675 DESCRIPTION

31676 Refer to *pthread_attr_destroy()*.

31677 **NAME**

31678 pthread_attr_setdetachstate — set the detachstate attribute

31679 **SYNOPSIS**31680 THR `#include <pthread.h>`31681 `int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);`

31682

31683 **DESCRIPTION**31684 Refer to *pthread_attr_getdetachstate()*.

31685 NAME

31686 pthread_attr_setguardsize — set the thread guardsize attribute

31687 SYNOPSIS

31688 XSI `#include <pthread.h>`

```
31689 int pthread_attr_setguardsize(pthread_attr_t *attr,  
31690                             size_t guardsize);
```

31691

31692 DESCRIPTION

31693 Refer to *pthread_attr_getguardsize()*.

31694 **NAME**31695 pthread_attr_setinheritsched — set the inheritsched attribute (**REALTIME THREADS**)31696 **SYNOPSIS**31697 THR TPS `#include <pthread.h>`31698 `int pthread_attr_setinheritsched(pthread_attr_t *attr,`
31699 `int inheritsched);`

31700

31701 **DESCRIPTION**31702 Refer to *pthread_attr_getinheritsched()*.

31703 NAME

31704 pthread_attr_setschedparam — set the schedparam attribute

31705 SYNOPSIS

31706 THR `#include <pthread.h>`

```
31707 int pthread_attr_setschedparam(pthread_attr_t *restrict attr,  
31708     const struct sched_param *restrict param);
```

31709

31710 DESCRIPTION

31711 Refer to *pthread_attr_getschedparam()*.

31712 **NAME**31713 pthread_attr_setschedpolicy — set the schedpolicy attribute (**REALTIME THREADS**)31714 **SYNOPSIS**31715 THR TPS `#include <pthread.h>`31716 `int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);`

31717

31718 **DESCRIPTION**31719 Refer to *pthread_attr_getschedpolicy()*.

31720 NAME

31721 pthread_attr_setscope — set the contention scope attribute (**REALTIME THREADS**)

31722 SYNOPSIS

31723 THR TPS `#include <pthread.h>`

31724 `int pthread_attr_setscope(pthread_attr_t *attr, int contention_scope);`

31725

31726 DESCRIPTION

31727 Refer to *pthread_attr_getscope()*.

31728 **NAME**

31729 pthread_attr_setstack — set the stack attribute

31730 **SYNOPSIS**31731 XSI `#include <pthread.h>`31732 `int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr,`
31733 `size_t stacksize);`

31734

31735 **DESCRIPTION**31736 Refer to *pthread_attr_getstack()*.

31737 NAME

31738 pthread_attr_setstackaddr — set the stackaddr attribute

31739 SYNOPSIS

31740 THR TSA `#include <pthread.h>`

31741 OB `int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);`

31742

31743 DESCRIPTION

31744 Refer to *pthread_attr_getstackaddr()*.

31745 **NAME**

31746 pthread_attr_setstacksize — set the stacksize attribute

31747 **SYNOPSIS**31748 THR TSA `#include <pthread.h>`31749 `int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);`

31750

31751 **DESCRIPTION**31752 Refer to *pthread_attr_getstacksize()*.

31753 NAME

31754 pthread_barrier_destroy, pthread_barrier_init — destroy and initialize a barrier object
 31755 (ADVANCED REALTIME THREADS)

31756 SYNOPSIS

31757 THR BAR #include <pthread.h>

```
31758 int pthread_barrier_destroy(pthread_barrier_t *barrier);
31759 int pthread_barrier_init(pthread_barrier_t *restrict barrier,
31760 const pthread_barrierattr_t *restrict attr, unsigned count);
31761
```

31762 DESCRIPTION

31763 The *pthread_barrier_destroy()* function shall destroy the barrier referenced by *barrier* and release
 31764 any resources used by the barrier. The effect of subsequent use of the barrier is undefined until
 31765 the barrier is reinitialized by another call to *pthread_barrier_init()*. An implementation may use
 31766 this function to set *barrier* to an invalid value. The results are undefined if
 31767 *pthread_barrier_destroy()* is called when any thread is blocked on the barrier, or if this function is
 31768 called with an uninitialized barrier.

31769 The *pthread_barrier_init()* function shall allocate any resources required to use the barrier
 31770 referenced by *barrier* and shall initialize the barrier with attributes referenced by *attr*. If *attr* is
 31771 NULL, the default barrier attributes shall be used; the effect is the same as passing the address of
 31772 a default barrier attributes object. The results are undefined if *pthread_barrier_init()* is called
 31773 when any thread is blocked on the barrier (that is, has not returned from the
 31774 *pthread_barrier_wait()* call). The results are undefined if a barrier is used without first being
 31775 initialized. The results are undefined if *pthread_barrier_init()* is called specifying an already
 31776 initialized barrier.

31777 The *count* argument specifies the number of threads that must call *pthread_barrier_wait()* before
 31778 any of them successfully return from the call. The value specified by *count* must be greater than
 31779 zero.

31780 If the *pthread_barrier_init()* function fails, the barrier shall not be initialized and the contents of
 31781 *barrier* are undefined.

31782 Only the object referenced by *barrier* may be used for performing synchronization. The result of
 31783 referring to copies of that object in calls to *pthread_barrier_destroy()* or *pthread_barrier_wait()* is
 31784 undefined.

31785 RETURN VALUE

31786 Upon successful completion, these functions shall return zero; otherwise, an error number shall
 31787 be returned to indicate the error.

31788 ERRORS

31789 The *pthread_barrier_destroy()* function may fail if:

31790 [EBUSY] The implementation has detected an attempt to destroy a barrier while it is in
 31791 use (for example, while being used in a *pthread_barrier_wait()* call) by another
 31792 thread.

31793 [EINVAL] The value specified by *barrier* is invalid.

31794 The *pthread_barrier_init()* function shall fail if:

31795 [EAGAIN] The system lacks the necessary resources to initialize another barrier.

31796 [EINVAL] The value specified by *count* is equal to zero.

- 31797 [ENOMEM] Insufficient memory exists to initialize the barrier.
- 31798 The *pthread_barrier_init()* function may fail if:
- 31799 [EBUSY] The implementation has detected an attempt to reinitialize a barrier while it is
31800 in use (for example, while being used in a *pthread_barrier_wait()* call) by
31801 another thread.
- 31802 [EINVAL] The value specified by *attr* is invalid.
- 31803 These functions shall not return an error code of [EINTR].
- 31804 **EXAMPLES**
- 31805 None.
- 31806 **APPLICATION USAGE**
- 31807 The *pthread_barrier_destroy()* and *pthread_barrier_init()* functions are part of the Barriers option
31808 and need not be provided on all implementations.
- 31809 **RATIONALE**
- 31810 None.
- 31811 **FUTURE DIRECTIONS**
- 31812 None.
- 31813 **SEE ALSO**
- 31814 *pthread_barrier_wait()*, the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>
- 31815 **CHANGE HISTORY**
- 31816 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

31817 **NAME**31818 pthread_barrier_wait — synchronize at a barrier (**ADVANCED REALTIME THREADS**)31819 **SYNOPSIS**

31820 THR BAR #include <pthread.h>

31821 int pthread_barrier_wait(pthread_barrier_t *barrier);

31822

31823 **DESCRIPTION**

31824 The *pthread_barrier_wait()* function shall synchronize participating threads at the barrier
 31825 referenced by *barrier*. The calling thread shall block until the required number of threads have
 31826 called *pthread_barrier_wait()* specifying the barrier.

31827 When the required number of threads have called *pthread_barrier_wait()* specifying the barrier,
 31828 the constant PTHREAD_BARRIER_SERIAL_THREAD shall be returned to one unspecified
 31829 thread and zero shall be returned to each of the remaining threads. At this point, the barrier shall
 31830 be reset to the state it had as a result of the most recent *pthread_barrier_init()* function that
 31831 referenced it.

31832 The constant PTHREAD_BARRIER_SERIAL_THREAD is defined in <pthread.h> and its value
 31833 shall be distinct from any other value returned by *pthread_barrier_wait()*.

31834 The results are undefined if this function is called with an uninitialized barrier.

31835 If a signal is delivered to a thread blocked on a barrier, upon return from the signal handler the
 31836 thread shall resume waiting at the barrier if the barrier wait has not completed (that is, if the
 31837 required number of threads have not arrived at the barrier during the execution of the signal
 31838 handler); otherwise, the thread shall continue as normal from the completed barrier wait. Until
 31839 the thread in the signal handler returns from it, it is unspecified whether other threads may
 31840 proceed past the barrier once they have all reached it.

31841 A thread that has blocked on a barrier shall not prevent any unblocked thread that is eligible to
 31842 use the same processing resources from eventually making forward progress in its execution.
 31843 Eligibility for processing resources shall be determined by the scheduling policy.

31844 **RETURN VALUE**

31845 Upon successful completion, the *pthread_barrier_wait()* function shall return
 31846 PTHREAD_BARRIER_SERIAL_THREAD for a single (arbitrary) thread synchronized at the
 31847 barrier and zero for each of the other threads. Otherwise, an error number shall be returned to
 31848 indicate the error.

31849 **ERRORS**

31850 The *pthread_barrier_wait()* function may fail if:

31851 [EINVAL] The value specified by *barrier* does not refer to an initialized barrier object.

31852 This function shall not return an error code of [EINTR].

31853 **EXAMPLES**

31854 None.

31855 **APPLICATION USAGE**

31856 Applications using this function may be subject to priority inversion, as discussed in the Base
 31857 Definitions volume of IEEE Std 1003.1-2001, Section 3.285, Priority Inversion.

31858 The *pthread_barrier_wait()* function is part of the Barriers option and need not be provided on all
 31859 implementations.

31860 **RATIONALE**

31861 None.

31862 **FUTURE DIRECTIONS**

31863 None.

31864 **SEE ALSO**31865 *pthread_barrier_destroy()*, the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>31866 **CHANGE HISTORY**

31867 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

31868 In the SYNOPSIS, the inclusion of <sys/types.h> is no longer required.

31869 **NAME**

31870 pthread_barrierattr_destroy, pthread_barrierattr_init — destroy and initialize the barrier
31871 attributes object (**ADVANCED REALTIME THREADS**)

31872 **SYNOPSIS**

31873 THR BAR #include <pthread.h>

31874 int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);

31875 int pthread_barrierattr_init(pthread_barrierattr_t *attr);

31876

31877 **DESCRIPTION**

31878 The *pthread_barrierattr_destroy()* function shall destroy a barrier attributes object. A destroyed
31879 *attr* attributes object can be reinitialized using *pthread_barrierattr_init()*; the results of otherwise
31880 referencing the object after it has been destroyed are undefined. An implementation may cause
31881 *pthread_barrierattr_destroy()* to set the object referenced by *attr* to an invalid value.

31882 The *pthread_barrierattr_init()* function shall initialize a barrier attributes object *attr* with the
31883 default value for all of the attributes defined by the implementation.

31884 Results are undefined if *pthread_barrierattr_init()* is called specifying an already initialized *attr*
31885 attributes object.

31886 After a barrier attributes object has been used to initialize one or more barriers, any function
31887 affecting the attributes object (including destruction) shall not affect any previously initialized
31888 barrier.

31889 **RETURN VALUE**

31890 If successful, the *pthread_barrierattr_destroy()* and *pthread_barrierattr_init()* functions shall return
31891 zero; otherwise, an error number shall be returned to indicate the error.

31892 **ERRORS**

31893 The *pthread_barrierattr_destroy()* function may fail if:

31894 [EINVAL] The value specified by *attr* is invalid.

31895 The *pthread_barrierattr_init()* function shall fail if:

31896 [ENOMEM] Insufficient memory exists to initialize the barrier attributes object.

31897 These functions shall not return an error code of [EINTR].

31898 **EXAMPLES**

31899 None.

31900 **APPLICATION USAGE**

31901 The *pthread_barrierattr_destroy()* and *pthread_barrierattr_init()* functions are part of the Barriers
31902 option and need not be provided on all implementations.

31903 **RATIONALE**

31904 None.

31905 **FUTURE DIRECTIONS**

31906 None.

31907 **SEE ALSO**

31908 *pthread_barrierattr_getpshared()*, *pthread_barrierattr_setpshared()*, the Base Definitions volume of
31909 IEEE Std 1003.1-2001, <pthread.h>.

31910 **CHANGE HISTORY**

- 31911 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.
- 31912 In the SYNOPSIS, the inclusion of `<sys/types.h>` is no longer required.

31913 **NAME**

31914 pthread_barrierattr_getpshared, pthread_barrierattr_setpshared — get and set the process-
 31915 shared attribute of the barrier attributes object (**ADVANCED REALTIME THREADS**)

31916 **SYNOPSIS**

31917 THR `#include <pthread.h>`

```
31918 BAR TSH int pthread_barrierattr_getpshared(const pthread_barrierattr_t *
31919          restrict attr, int *restrict pshared);
31920 int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,
31921          int pshared);
31922
```

31923 **DESCRIPTION**

31924 The *pthread_barrierattr_getpshared()* function shall obtain the value of the *process-shared* attribute
 31925 from the attributes object referenced by *attr*. The *pthread_barrierattr_setpshared()* function shall
 31926 set the *process-shared* attribute in an initialized attributes object referenced by *attr*.

31927 The *process-shared* attribute is set to PTHREAD_PROCESS_SHARED to permit a barrier to be
 31928 operated upon by any thread that has access to the memory where the barrier is allocated. If the
 31929 *process-shared* attribute is PTHREAD_PROCESS_PRIVATE, the barrier shall only be operated
 31930 upon by threads created within the same process as the thread that initialized the barrier; if
 31931 threads of different processes attempt to operate on such a barrier, the behavior is undefined.
 31932 The default value of the attribute shall be PTHREAD_PROCESS_PRIVATE. Both constants
 31933 PTHREAD_PROCESS_SHARED and PTHREAD_PROCESS_PRIVATE are defined in
 31934 `<pthread.h>`.

31935 Additional attributes, their default values, and the names of the associated functions to get and
 31936 set those attribute values are implementation-defined.

31937 **RETURN VALUE**

31938 If successful, the *pthread_barrierattr_getpshared()* function shall return zero and store the value of
 31939 the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise,
 31940 an error number shall be returned to indicate the error.

31941 If successful, the *pthread_barrierattr_setpshared()* function shall return zero; otherwise, an error
 31942 number shall be returned to indicate the error.

31943 **ERRORS**

31944 These functions may fail if:

31945 [EINVAL] The value specified by *attr* is invalid.

31946 The *pthread_barrierattr_setpshared()* function may fail if:

31947 [EINVAL] The new value specified for the *process-shared* attribute is not one of the legal
 31948 values PTHREAD_PROCESS_SHARED or PTHREAD_PROCESS_PRIVATE.

31949 These functions shall not return an error code of [EINTR].

31950 **EXAMPLES**

31951 None.

31952 **APPLICATION USAGE**

31953 The *pthread_barrierattr_getpshared()* and *pthread_barrierattr_setpshared()* functions are part of the
31954 Barriers option and need not be provided on all implementations.

31955 **RATIONALE**

31956 None.

31957 **FUTURE DIRECTIONS**

31958 None.

31959 **SEE ALSO**

31960 *pthread_barrier_destroy()*, *pthread_barrierattr_destroy()*, *pthread_barrierattr_init()*, the Base
31961 Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

31962 **CHANGE HISTORY**

31963 First released in Issue 6. Derived from IEEE Std 1003.1j-2000

31964 NAME

31965 pthread_barrierattr_init — initialize the barrier attributes object (**ADVANCED REALTIME**
31966 **THREADS**)

31967 SYNOPSIS

31968 THR BAR #include <pthread.h>

31969 int pthread_barrierattr_init(pthread_barrierattr_t *attr);

31970

31971 DESCRIPTION

31972 Refer to *pthread_barrierattr_destroy()*.

31973 **NAME**

31974 pthread_barrierattr_setpshared — set the process-shared attribute of the barrier attributes object
31975 (**ADVANCED REALTIME THREADS**)

31976 **SYNOPSIS**

31977 THR #include <pthread.h>

31978 BAR TSH int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,
31979 int pshared);

31980

31981 **DESCRIPTION**

31982 Refer to *pthread_barrierattr_getpshared()*.

31983 **NAME**

31984 pthread_cancel — cancel execution of a thread

31985 **SYNOPSIS**

31986 THR #include <pthread.h>

31987 int pthread_cancel(pthread_t thread);

31988

31989 **DESCRIPTION**

31990 The *pthread_cancel()* function shall request that *thread* be canceled. The target thread's
 31991 cancelability state and type determines when the cancelation takes effect. When the cancelation
 31992 is acted on, the cancelation cleanup handlers for *thread* shall be called. When the last cancelation
 31993 cleanup handler returns, the thread-specific data destructor functions shall be called for *thread*.
 31994 When the last destructor function returns, *thread* shall be terminated.

31995 The cancelation processing in the target thread shall run asynchronously with respect to the
 31996 calling thread returning from *pthread_cancel()*.

31997 **RETURN VALUE**

31998 If successful, the *pthread_cancel()* function shall return zero; otherwise, an error number shall be
 31999 returned to indicate the error.

32000 **ERRORS**32001 The *pthread_cancel()* function may fail if:

32002 [ESRCH] No thread could be found corresponding to that specified by the given thread
 32003 ID.

32004 The *pthread_cancel()* function shall not return an error code of [EINTR].32005 **EXAMPLES**

32006 None.

32007 **APPLICATION USAGE**

32008 None.

32009 **RATIONALE**

32010 Two alternative functions were considered for sending the cancelation notification to a thread.
 32011 One would be to define a new SIGCANCEL signal that had the cancelation semantics when
 32012 delivered; the other was to define the new *pthread_cancel()* function, which would trigger the
 32013 cancelation semantics.

32014 The advantage of a new signal was that so much of the delivery criteria were identical to that
 32015 used when trying to deliver a signal that making cancelation notification a signal was seen as
 32016 consistent. Indeed, many implementations implement cancelation using a special signal. On the
 32017 other hand, there would be no signal functions that could be used with this signal except
 32018 *pthread_kill()*, and the behavior of the delivered cancelation signal would be unlike any
 32019 previously existing defined signal.

32020 The benefits of a special function include the recognition that this signal would be defined
 32021 because of the similar delivery criteria and that this is the only common behavior between a
 32022 cancelation request and a signal. In addition, the cancelation delivery mechanism does not have
 32023 to be implemented as a signal. There are also strong, if not stronger, parallels with language
 32024 exception mechanisms than with signals that are potentially obscured if the delivery mechanism
 32025 is visibly closer to signals.

32026 In the end, it was considered that as there were so many exceptions to the use of the new signal
 32027 with existing signals functions it would be misleading. A special function has resolved this

32028 problem. This function was carefully defined so that an implementation wishing to provide the
32029 cancelation functions on top of signals could do so. The special function also means that
32030 implementations are not obliged to implement cancelation with signals.

32031 **FUTURE DIRECTIONS**

32032 None.

32033 **SEE ALSO**

32034 *pthread_exit()*, *pthread_cond_timedwait()*, *pthread_join()*, *pthread_setcancelstate()*, the Base
32035 Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

32036 **CHANGE HISTORY**

32037 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

32038 **Issue 6**

32039 The *pthread_cancel()* function is marked as part of the Threads option.

32040 **NAME**

32041 pthread_cleanup_pop, pthread_cleanup_push — establish cancelation handlers

32042 **SYNOPSIS**

32043 THR #include <pthread.h>

32044 void pthread_cleanup_pop(int execute);

32045 void pthread_cleanup_push(void (*routine)(void*), void *arg);

32046

32047 **DESCRIPTION**32048 The *pthread_cleanup_pop()* function shall remove the routine at the top of the calling thread's
32049 cancelation cleanup stack and optionally invoke it (if *execute* is non-zero).32050 The *pthread_cleanup_push()* function shall push the specified cancelation cleanup handler *routine*
32051 onto the calling thread's cancelation cleanup stack. The cancelation cleanup handler shall be
32052 popped from the cancelation cleanup stack and invoked with the argument *arg* when:

- 32053 • The thread exits (that is, calls *pthread_exit()*).
- 32054 • The thread acts upon a cancelation request.
- 32055 • The thread calls *pthread_cleanup_pop()* with a non-zero *execute* argument.

32056 These functions may be implemented as macros. The application shall ensure that they appear
32057 as statements, and in pairs within the same lexical scope (that is, the *pthread_cleanup_push()*
32058 macro may be thought to expand to a token list whose first token is '{' with
32059 *pthread_cleanup_pop()* expanding to a token list whose last token is the corresponding '}').32060 The effect of calling *longjmp()* or *siglongjmp()* is undefined if there have been any calls to
32061 *pthread_cleanup_push()* or *pthread_cleanup_pop()* made without the matching call since the jump
32062 buffer was filled. The effect of calling *longjmp()* or *siglongjmp()* from inside a cancelation
32063 cleanup handler is also undefined unless the jump buffer was also filled in the cancelation
32064 cleanup handler.32065 **RETURN VALUE**32066 The *pthread_cleanup_push()* and *pthread_cleanup_pop()* functions shall not return a value.32067 **ERRORS**

32068 No errors are defined.

32069 These functions shall not return an error code of [EINTR].

32070 **EXAMPLES**32071 The following is an example using thread primitives to implement a cancelable, writers-priority
32072 read-write lock:

```

32073 typedef struct {
32074     pthread_mutex_t lock;
32075     pthread_cond_t rcond,
32076     wcond;
32077     int lock_count; /* < 0 .. Held by writer. */
32078                   /* > 0 .. Held by lock_count readers. */
32079                   /* = 0 .. Held by nobody. */
32080     int waiting_writers; /* Count of waiting writers. */
32081 } rwlock;

32082 void
32083 waiting_reader_cleanup(void *arg)
32084 {

```



```

32085         rwlock *l;
32086         l = (rwlock *) arg;
32087         pthread_mutex_unlock(&l->lock);
32088     }
32089
32089 void
32090 lock_for_read(rwlock *l)
32091 {
32092     pthread_mutex_lock(&l->lock);
32093     pthread_cleanup_push(waiting_reader_cleanup, l);
32094     while ((l->lock_count < 0) && (l->waiting_writers != 0))
32095         pthread_cond_wait(&l->rcond, &l->lock);
32096     l->lock_count++;
32097     /*
32098      * Note the pthread_cleanup_pop executes
32099      * waiting_reader_cleanup.
32100      */
32101     pthread_cleanup_pop(1);
32102 }
32103
32103 void
32104 release_read_lock(rwlock *l)
32105 {
32106     pthread_mutex_lock(&l->lock);
32107     if (--l->lock_count == 0)
32108         pthread_cond_signal(&l->wcond);
32109     pthread_mutex_unlock(l);
32110 }
32111
32111 void
32112 waiting_writer_cleanup(void *arg)
32113 {
32114     rwlock *l;
32115
32116     l = (rwlock *) arg;
32117     if ((--l->waiting_writers == 0) && (l->lock_count >= 0)) {
32118         /*
32119          * This only happens if we have been canceled.
32120          */
32121         pthread_cond_broadcast(&l->wcond);
32122     }
32123     pthread_mutex_unlock(&l->lock);
32124 }
32125
32125 void
32126 lock_for_write(rwlock *l)
32127 {
32128     pthread_mutex_lock(&l->lock);
32129     l->waiting_writers++;
32130     pthread_cleanup_push(waiting_writer_cleanup, l);
32131     while (l->lock_count != 0)
32132         pthread_cond_wait(&l->wcond, &l->lock);
32133     l->lock_count = -1;
32134     /*

```



```
32134         * Note the pthread_cleanup_pop executes
32135         * waiting_writer_cleanup.
32136         */
32137         pthread_cleanup_pop(1);
32138     }
32139
32139 void
32140 release_write_lock(rwlock *l)
32141 {
32142     pthread_mutex_lock(&l->lock);
32143     l->lock_count = 0;
32144     if (l->waiting_writers == 0)
32145         pthread_cond_broadcast(&l->rcond)
32146     else
32147         pthread_cond_signal(&l->wcond);
32148     pthread_mutex_unlock(&l->lock);
32149 }
32150
32150 /*
32151  * This function is called to initialize the read/write lock.
32152  */
32153 void
32154 initialize_rwlock(rwlock *l)
32155 {
32156     pthread_mutex_init(&l->lock, pthread_mutexattr_default);
32157     pthread_cond_init(&l->wcond, pthread_condattr_default);
32158     pthread_cond_init(&l->rcond, pthread_condattr_default);
32159     l->lock_count = 0;
32160     l->waiting_writers = 0;
32161 }
32162
32162 reader_thread()
32163 {
32164     lock_for_read(&lock);
32165     pthread_cleanup_push(release_read_lock, &lock);
32166     /*
32167      * Thread has read lock.
32168      */
32169     pthread_cleanup_pop(1);
32170 }
32171
32171 writer_thread()
32172 {
32173     lock_for_write(&lock);
32174     pthread_cleanup_push(release_write_lock, &lock);
32175     /*
32176      * Thread has write lock.
32177      */
32178     pthread_cleanup_pop(1);
32179 }
```


32180 **APPLICATION USAGE**

32181 The two routines that push and pop cancelation cleanup handlers, *pthread_cleanup_push()* and
 32182 *pthread_cleanup_pop()*, can be thought of as left and right parentheses. They always need to be
 32183 matched.

32184 **RATIONALE**

32185 The restriction that the two routines that push and pop cancelation cleanup handlers,
 32186 *pthread_cleanup_push()* and *pthread_cleanup_pop()*, have to appear in the same lexical scope
 32187 allows for efficient macro or compiler implementations and efficient storage management. A
 32188 sample implementation of these routines as macros might look like this:

```
32189 #define pthread_cleanup_push(rtn,arg) { \
32190     struct _pthread_handler_rec __cleanup_handler, *__head; \
32191     __cleanup_handler.rtn = rtn; \
32192     __cleanup_handler.arg = arg; \
32193     (void) pthread_getspecific(_pthread_handler_key, &__head); \
32194     __cleanup_handler.next = *__head; \
32195     *__head = &__cleanup_handler;
32196
32197 #define pthread_cleanup_pop(ex) \
32198     *__head = __cleanup_handler.next; \
32199     if (ex) (*__cleanup_handler.rtn)(__cleanup_handler.arg); \
32199 }
```

32200 A more ambitious implementation of these routines might do even better by allowing the
 32201 compiler to note that the cancelation cleanup handler is a constant and can be expanded inline.

32202 This volume of IEEE Std 1003.1-2001 currently leaves unspecified the effect of calling *longjmp()*
 32203 from a signal handler executing in a POSIX System Interfaces function. If an implementation
 32204 wants to allow this and give the programmer reasonable behavior, the *longjmp()* function has to
 32205 call all cancelation cleanup handlers that have been pushed but not popped since the time
 32206 *setjmp()* was called.

32207 Consider a multi-threaded function called by a thread that uses signals. If a signal were
 32208 delivered to a signal handler during the operation of *qsort()* and that handler were to call
 32209 *longjmp()* (which, in turn, did *not* call the cancelation cleanup handlers) the helper threads
 32210 created by the *qsort()* function would not be canceled. Instead, they would continue to execute
 32211 and write into the argument array even though the array might have been popped off the stack.

32212 Note that the specified cleanup handling mechanism is especially tied to the C language and,
 32213 while the requirement for a uniform mechanism for expressing cleanup is language-
 32214 independent, the mechanism used in other languages may be quite different. In addition, this
 32215 mechanism is really only necessary due to the lack of a real exception mechanism in the C
 32216 language, which would be the ideal solution.

32217 There is no notion of a cancelation cleanup-safe function. If an application has no cancelation
 32218 points in its signal handlers, blocks any signal whose handler may have cancelation points while
 32219 calling async-unsafe functions, or disables cancelation while calling async-unsafe functions, all
 32220 functions may be safely called from cancelation cleanup routines.

32221 **FUTURE DIRECTIONS**

32222 None.

32223 **SEE ALSO**

32224 *pthread_cancel()*, *pthread_setcancelstate()*, the Base Definitions volume of IEEE Std 1003.1-2001,
 32225 <pthread.h>

32226 **CHANGE HISTORY**

32227 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

32228 **Issue 6**

32229 The *pthread_cleanup_pop()* and *pthread_cleanup_push()* functions are marked as part of the
32230 Threads option.

32231 The APPLICATION USAGE section is added.

32232 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

32233 **NAME**

32234 pthread_cond_broadcast, pthread_cond_signal — broadcast or signal a condition

32235 **SYNOPSIS**

32236 THR #include <pthread.h>

32237 int pthread_cond_broadcast(pthread_cond_t *cond);

32238 int pthread_cond_signal(pthread_cond_t *cond);

32239

32240 **DESCRIPTION**

32241 These functions shall unblock threads blocked on a condition variable.

32242 The *pthread_cond_broadcast()* function shall unblock all threads currently blocked on the
32243 specified condition variable *cond*.32244 The *pthread_cond_signal()* function shall unblock at least one of the threads that are blocked on
32245 the specified condition variable *cond* (if any threads are blocked on *cond*).32246 If more than one thread is blocked on a condition variable, the scheduling policy shall determine
32247 the order in which threads are unblocked. When each thread unblocked as a result of a
32248 *pthread_cond_broadcast()* or *pthread_cond_signal()* returns from its call to *pthread_cond_wait()* or
32249 *pthread_cond_timedwait()*, the thread shall own the mutex with which it called
32250 *pthread_cond_wait()* or *pthread_cond_timedwait()*. The thread(s) that are unblocked shall contend
32251 for the mutex according to the scheduling policy (if applicable), and as if each had called
32252 *pthread_mutex_lock()*.32253 The *pthread_cond_broadcast()* or *pthread_cond_signal()* functions may be called by a thread
32254 whether or not it currently owns the mutex that threads calling *pthread_cond_wait()* or
32255 *pthread_cond_timedwait()* have associated with the condition variable during their waits;
32256 however, if predictable scheduling behavior is required, then that mutex shall be locked by the
32257 thread calling *pthread_cond_broadcast()* or *pthread_cond_signal()*.32258 The *pthread_cond_broadcast()* and *pthread_cond_signal()* functions shall have no effect if there are
32259 no threads currently blocked on *cond*.32260 **RETURN VALUE**32261 If successful, the *pthread_cond_broadcast()* and *pthread_cond_signal()* functions shall return zero;
32262 otherwise, an error number shall be returned to indicate the error.32263 **ERRORS**32264 The *pthread_cond_broadcast()* and *pthread_cond_signal()* function may fail if:32265 [EINVAL] The value *cond* does not refer to an initialized condition variable.

32266 These functions shall not return an error code of [EINTR].

32267 **EXAMPLES**

32268 None.

32269 **APPLICATION USAGE**32270 The *pthread_cond_broadcast()* function is used whenever the shared-variable state has been
32271 changed in a way that more than one thread can proceed with its task. Consider a single
32272 producer/multiple consumer problem, where the producer can insert multiple items on a list
32273 that is accessed one item at a time by the consumers. By calling the *pthread_cond_broadcast()*
32274 function, the producer would notify all consumers that might be waiting, and thereby the
32275 application would receive more throughput on a multi-processor. In addition,
32276 *pthread_cond_broadcast()* makes it easier to implement a read-write lock. The
32277 *pthread_cond_broadcast()* function is needed in order to wake up all waiting readers when a

writer releases its lock. Finally, the two-phase commit algorithm can use this broadcast function to notify all clients of an impending transaction commit.

It is not safe to use the *pthread_cond_signal()* function in a signal handler that is invoked asynchronously. Even if it were safe, there would still be a race between the test of the Boolean *pthread_cond_wait()* that could not be efficiently eliminated.

Mutexes and condition variables are thus not suitable for releasing a waiting thread by signaling from code running in a signal handler.

RATIONALE

Multiple Awakenings by Condition Signal

On a multi-processor, it may be impossible for an implementation of *pthread_cond_signal()* to avoid the unblocking of more than one thread blocked on a condition variable. For example, consider the following partial implementation of *pthread_cond_wait()* and *pthread_cond_signal()*, executed by two threads in the order given. One thread is trying to wait on the condition variable, another is concurrently executing *pthread_cond_signal()*, while a third thread is already waiting.

```
pthread_cond_wait(mutex, cond):
    value = cond->value; /* 1 */
    pthread_mutex_unlock(mutex); /* 2 */
    pthread_mutex_lock(cond->mutex); /* 10 */
    if (value == cond->value) { /* 11 */
        me->next_cond = cond->waiter;
        cond->waiter = me;
        pthread_mutex_unlock(cond->mutex);
        unable_to_run(me);
    } else
        pthread_mutex_unlock(cond->mutex); /* 12 */
    pthread_mutex_lock(mutex); /* 13 */

pthread_cond_signal(cond):
    pthread_mutex_lock(cond->mutex); /* 3 */
    cond->value++; /* 4 */
    if (cond->waiter) { /* 5 */
        sleeper = cond->waiter; /* 6 */
        cond->waiter = sleeper->next_cond; /* 7 */
        able_to_run(sleeper); /* 8 */
    }
    pthread_mutex_unlock(cond->mutex); /* 9 */
```

The effect is that more than one thread can return from its call to *pthread_cond_wait()* or *pthread_cond_timedwait()* as a result of one call to *pthread_cond_signal()*. This effect is called “spurious wakeup”. Note that the situation is self-correcting in that the number of threads that are so awakened is finite; for example, the next thread to call *pthread_cond_wait()* after the sequence of events above blocks.

While this problem could be resolved, the loss of efficiency for a fringe condition that occurs only rarely is unacceptable, especially given that one has to check the predicate associated with a condition variable anyway. Correcting this problem would unnecessarily reduce the degree of concurrency in this basic building block for all higher-level synchronization operations.

An added benefit of allowing spurious wakeups is that applications are forced to code a predicate-testing-loop around the condition wait. This also makes the application tolerate

32325 superfluous condition broadcasts or signals on the same condition variable that may be coded in
32326 some other part of the application. The resulting applications are thus more robust. Therefore,
32327 IEEE Std 1003.1-2001 explicitly documents that spurious wakeups may occur.

32328 **FUTURE DIRECTIONS**

32329 None.

32330 **SEE ALSO**

32331 *pthread_cond_destroy()*, *pthread_cond_timedwait()*, the Base Definitions volume of
32332 IEEE Std 1003.1-2001, <**pthread.h**>

32333 **CHANGE HISTORY**

32334 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

32335 **Issue 6**

32336 The *pthread_cond_broadcast()* and *pthread_cond_signal()* functions are marked as part of the
32337 Threads option.

32338 The APPLICATION USAGE section is added.

32339 NAME

32340 pthread_cond_destroy, pthread_cond_init — destroy and initialize condition variables

32341 SYNOPSIS

32342 THR #include <pthread.h>

```

32343 int pthread_cond_destroy(pthread_cond_t *cond);
32344 int pthread_cond_init(pthread_cond_t *restrict cond,
32345     const pthread_condattr_t *restrict attr);
32346 pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
32347

```

32348 DESCRIPTION

32349 The *pthread_cond_destroy()* function shall destroy the given condition variable specified by *cond*;
 32350 the object becomes, in effect, uninitialized. An implementation may cause *pthread_cond_destroy()*
 32351 to set the object referenced by *cond* to an invalid value. A destroyed condition variable object can
 32352 be reinitialized using *pthread_cond_init()*; the results of otherwise referencing the object after it
 32353 has been destroyed are undefined.

32354 It shall be safe to destroy an initialized condition variable upon which no threads are currently
 32355 blocked. Attempting to destroy a condition variable upon which other threads are currently
 32356 blocked results in undefined behavior.

32357 The *pthread_cond_init()* function shall initialize the condition variable referenced by *cond* with
 32358 attributes referenced by *attr*. If *attr* is NULL, the default condition variable attributes shall be
 32359 used; the effect is the same as passing the address of a default condition variable attributes
 32360 object. Upon successful initialization, the state of the condition variable shall become initialized.

32361 Only *cond* itself may be used for performing synchronization. The result of referring to copies of
 32362 *cond* in calls to *pthread_cond_wait()*, *pthread_cond_timedwait()*, *pthread_cond_signal()*,
 32363 *pthread_cond_broadcast()*, and *pthread_cond_destroy()* is undefined.

32364 Attempting to initialize an already initialized condition variable results in undefined behavior.

32365 In cases where default condition variable attributes are appropriate, the macro
 32366 PTHREAD_COND_INITIALIZER can be used to initialize condition variables that are statically
 32367 allocated. The effect shall be equivalent to dynamic initialization by a call to *pthread_cond_init()*
 32368 with parameter *attr* specified as NULL, except that no error checks are performed.

32369 RETURN VALUE

32370 If successful, the *pthread_cond_destroy()* and *pthread_cond_init()* functions shall return zero;
 32371 otherwise, an error number shall be returned to indicate the error.

32372 The [EBUSY] and [EINVAL] error checks, if implemented, shall act as if they were performed
 32373 immediately at the beginning of processing for the function and caused an error return prior to
 32374 modifying the state of the condition variable specified by *cond*.

32375 ERRORS

32376 The *pthread_cond_destroy()* function may fail if:

32377 [EBUSY] The implementation has detected an attempt to destroy the object referenced
 32378 by *cond* while it is referenced (for example, while being used in a
 32379 *pthread_cond_wait()* or *pthread_cond_timedwait()*) by another thread.

32380 [EINVAL] The value specified by *cond* is invalid.

32381 The *pthread_cond_init()* function shall fail if:

32382 [EAGAIN] The system lacked the necessary resources (other than memory) to initialize
 32383 another condition variable.

32384 [ENOMEM] Insufficient memory exists to initialize the condition variable.

32385 The *pthread_cond_init()* function may fail if:

32386 [EBUSY] The implementation has detected an attempt to reinitialize the object
32387 referenced by *cond*, a previously initialized, but not yet destroyed, condition
32388 variable.

32389 [EINVAL] The value specified by *attr* is invalid.

32390 These functions shall not return an error code of [EINTR].

32391 EXAMPLES

32392 A condition variable can be destroyed immediately after all the threads that are blocked on it are
32393 awakened. For example, consider the following code:

```
32394 struct list {
32395     pthread_mutex_t lm;
32396     ...
32397 }
32398 struct elt {
32399     key k;
32400     int busy;
32401     pthread_cond_t notbusy;
32402     ...
32403 }
32404 /* Find a list element and reserve it. */
32405 struct elt *
32406 list_find(struct list *lp, key k)
32407 {
32408     struct elt *ep;
32409     pthread_mutex_lock(&lp->lm);
32410     while ((ep = find_elt(l, k) != NULL) && ep->busy)
32411         pthread_cond_wait(&ep->notbusy, &lp->lm);
32412     if (ep != NULL)
32413         ep->busy = 1;
32414     pthread_mutex_unlock(&lp->lm);
32415     return(ep);
32416 }
32417 delete_elt(struct list *lp, struct elt *ep)
32418 {
32419     pthread_mutex_lock(&lp->lm);
32420     assert(ep->busy);
32421     ... remove ep from list ...
32422     ep->busy = 0; /* Paranoid. */
32423     (A) pthread_cond_broadcast(&ep->notbusy);
32424     pthread_mutex_unlock(&lp->lm);
32425     (B) pthread_cond_destroy(&ep->notbusy);
32426     free(ep);
32427 }
```

32428 In this example, the condition variable and its list element may be freed (line B) immediately
32429 after all threads waiting for it are awakened (line A), since the mutex and the code ensure that no
32430 other thread can touch the element to be deleted.

32431 APPLICATION USAGE

32432 None.

32433 RATIONALE

32434 See *pthread_mutex_init()*; a similar rationale applies to condition variables.

32435 FUTURE DIRECTIONS

32436 None.

32437 SEE ALSO

32438 *pthread_cond_broadcast()*, *pthread_cond_signal()*, *pthread_cond_timedwait()*, the Base Definitions
32439 volume of IEEE Std 1003.1-2001, <**pthread.h**>

32440 CHANGE HISTORY

32441 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

32442 Issue 6

32443 The *pthread_cond_destroy()* and *pthread_cond_init()* functions are marked as part of the Threads
32444 option.

32445 IEEE PASC Interpretation 1003.1c #34 is applied, updating the DESCRIPTION.

32446 The **restrict** keyword is added to the *pthread_cond_init()* prototype for alignment with the
32447 ISO/IEC 9899:1999 standard.

32448 **NAME**

32449 pthread_cond_signal — signal a condition

32450 **SYNOPSIS**32451 THR `#include <pthread.h>`32452 `int pthread_cond_signal(pthread_cond_t *cond);`

32453

32454 **DESCRIPTION**32455 Refer to *pthread_cond_broadcast()*.

32456 NAME

32457 pthread_cond_timedwait, pthread_cond_wait — wait on a condition

32458 SYNOPSIS

32459 THR #include <pthread.h>

```

32460 int pthread_cond_timedwait(pthread_cond_t *restrict cond,
32461 pthread_mutex_t *restrict mutex,
32462 const struct timespec *restrict abstime);
32463 int pthread_cond_wait(pthread_cond_t *restrict cond,
32464 pthread_mutex_t *restrict mutex);
32465

```

32466 DESCRIPTION

32467 The *pthread_cond_timedwait()* and *pthread_cond_wait()* functions shall block on a condition
 32468 variable. They shall be called with *mutex* locked by the calling thread or undefined behavior
 32469 results.

32470 These functions atomically release *mutex* and cause the calling thread to block on the condition
 32471 variable *cond*; atomically here means “atomically with respect to access by another thread to the
 32472 mutex and then the condition variable”. That is, if another thread is able to acquire the mutex
 32473 after the about-to-block thread has released it, then a subsequent call to *pthread_cond_broadcast()*
 32474 or *pthread_cond_signal()* in that thread shall behave as if it were issued after the about-to-block
 32475 thread has blocked.

32476 Upon successful return, the mutex shall have been locked and shall be owned by the calling
 32477 thread.

32478 When using condition variables there is always a Boolean predicate involving shared variables
 32479 associated with each condition wait that is true if the thread should proceed. Spurious wakeups
 32480 from the *pthread_cond_timedwait()* or *pthread_cond_wait()* functions may occur. Since the return
 32481 from *pthread_cond_timedwait()* or *pthread_cond_wait()* does not imply anything about the value
 32482 of this predicate, the predicate should be re-evaluated upon such return.

32483 The effect of using more than one mutex for concurrent *pthread_cond_timedwait()* or
 32484 *pthread_cond_wait()* operations on the same condition variable is undefined; that is, a condition
 32485 variable becomes bound to a unique mutex when a thread waits on the condition variable, and
 32486 this (dynamic) binding shall end when the wait returns.

32487 A condition wait (whether timed or not) is a cancellation point. When the cancelability enable
 32488 state of a thread is set to PTHREAD_CANCEL_DEFERRED, a side effect of acting upon a
 32489 cancellation request while in a condition wait is that the mutex is (in effect) re-acquired before
 32490 calling the first cancellation cleanup handler. The effect is as if the thread were unblocked,
 32491 allowed to execute up to the point of returning from the call to *pthread_cond_timedwait()* or
 32492 *pthread_cond_wait()*, but at that point notices the cancellation request and instead of returning to
 32493 the caller of *pthread_cond_timedwait()* or *pthread_cond_wait()*, starts the thread cancellation
 32494 activities, which includes calling cancellation cleanup handlers.

32495 A thread that has been unblocked because it has been canceled while blocked in a call to
 32496 *pthread_cond_timedwait()* or *pthread_cond_wait()* shall not consume any condition signal that
 32497 may be directed concurrently at the condition variable if there are other threads blocked on the
 32498 condition variable.

32499 The *pthread_cond_timedwait()* function shall be equivalent to *pthread_cond_wait()*, except that an
 32500 error is returned if the absolute time specified by *abstime* passes (that is, system time equals or
 32501 exceeds *abstime*) before the condition *cond* is signaled or broadcasted, or if the absolute time
 32502 specified by *abstime* has already been passed at the time of the call.

32503 cs If the Clock Selection option is supported, the condition variable shall have a clock attribute
 32504 which specifies the clock that shall be used to measure the time specified by the *abstime*
 32505 argument. When such timeouts occur, *pthread_cond_timedwait()* shall nonetheless release and
 32506 re-acquire the mutex referenced by *mutex*. The *pthread_cond_timedwait()* function is also a
 32507 cancelation point.

32508 If a signal is delivered to a thread waiting for a condition variable, upon return from the signal
 32509 handler the thread resumes waiting for the condition variable as if it was not interrupted, or it
 32510 shall return zero due to spurious wakeup.

32511 RETURN VALUE

32512 Except in the case of [ETIMEDOUT], all these error checks shall act as if they were performed
 32513 immediately at the beginning of processing for the function and shall cause an error return, in
 32514 effect, prior to modifying the state of the mutex specified by *mutex* or the condition variable
 32515 specified by *cond*.

32516 Upon successful completion, a value of zero shall be returned; otherwise, an error number shall
 32517 be returned to indicate the error.

32518 ERRORS

32519 The *pthread_cond_timedwait()* function shall fail if:

32520 [ETIMEDOUT] The time specified by *abstime* to *pthread_cond_timedwait()* has passed.

32521 The *pthread_cond_timedwait()* and *pthread_cond_wait()* functions may fail if:

32522 [EINVAL] The value specified by *cond*, *mutex*, or *abstime* is invalid.

32523 [EINVAL] Different mutexes were supplied for concurrent *pthread_cond_timedwait()* or
 32524 *pthread_cond_wait()* operations on the same condition variable.

32525 [EPERM] The mutex was not owned by the current thread at the time of the call.

32526 These functions shall not return an error code of [EINTR].

32527 EXAMPLES

32528 None.

32529 APPLICATION USAGE

32530 None.

32531 RATIONALE

32532 Condition Wait Semantics

32533 It is important to note that when *pthread_cond_wait()* and *pthread_cond_timedwait()* return
 32534 without error, the associated predicate may still be false. Similarly, when
 32535 *pthread_cond_timedwait()* returns with the timeout error, the associated predicate may be true
 32536 due to an unavoidable race between the expiration of the timeout and the predicate state change.

32537 Some implementations, particularly on a multi-processor, may sometimes cause multiple
 32538 threads to wake up when the condition variable is signaled simultaneously on different
 32539 processors.

32540 In general, whenever a condition wait returns, the thread has to re-evaluate the predicate
 32541 associated with the condition wait to determine whether it can safely proceed, should wait
 32542 again, or should declare a timeout. A return from the wait does not imply that the associated
 32543 predicate is either true or false.

32544 It is thus recommended that a condition wait be enclosed in the equivalent of a “while loop”
 32545 that checks the predicate.

Timed Wait Semantics

An absolute time measure was chosen for specifying the timeout parameter for two reasons. First, a relative time measure can be easily implemented on top of a function that specifies absolute time, but there is a race condition associated with specifying an absolute timeout on top of a function that specifies relative timeouts. For example, assume that `clock_gettime()` returns the current time and `cond_relative_timed_wait()` uses relative timeouts:

```
clock_gettime(CLOCK_REALTIME, &now)
reltime = sleep_til_this_absolute_time - now;
cond_relative_timed_wait(c, m, &reltime);
```

If the thread is preempted between the first statement and the last statement, the thread blocks for too long. Blocking, however, is irrelevant if an absolute timeout is used. An absolute timeout also need not be recomputed if it is used multiple times in a loop, such as that enclosing a condition wait.

For cases when the system clock is advanced discontinuously by an operator, it is expected that implementations process any timed wait expiring at an intervening time as if that time had actually occurred.

Cancellation and Condition Wait

A condition wait, whether timed or not, is a cancellation point. That is, the functions `pthread_cond_wait()` or `pthread_cond_timedwait()` are points where a pending (or concurrent) cancellation request is noticed. The reason for this is that an indefinite wait is possible at these points—whatever event is being waited for, even if the program is totally correct, might never occur; for example, some input data being awaited might never be sent. By making condition wait a cancellation point, the thread can be canceled and perform its cancellation cleanup handler even though it may be stuck in some indefinite wait.

A side effect of acting on a cancellation request while a thread is blocked on a condition variable is to re-acquire the mutex before calling any of the cancellation cleanup handlers. This is done in order to ensure that the cancellation cleanup handler is executed in the same state as the critical code that lies both before and after the call to the condition wait function. This rule is also required when interfacing to POSIX threads from languages, such as Ada or C++, which may choose to map cancellation onto a language exception; this rule ensures that each exception handler guarding a critical section can always safely depend upon the fact that the associated mutex has already been locked regardless of exactly where within the critical section the exception was raised. Without this rule, there would not be a uniform rule that exception handlers could follow regarding the lock, and so coding would become very cumbersome.

Therefore, since *some* statement has to be made regarding the state of the lock when a cancellation is delivered during a wait, a definition has been chosen that makes application coding most convenient and error free.

When acting on a cancellation request while a thread is blocked on a condition variable, the implementation is required to ensure that the thread does not consume any condition signals directed at that condition variable if there are any other threads waiting on that condition variable. This rule is specified in order to avoid deadlock conditions that could occur if these two independent requests (one acting on a thread and the other acting on the condition variable) were not processed independently.

Performance of Mutexes and Condition Variables

Mutexes are expected to be locked only for a few instructions. This practice is almost automatically enforced by the desire of programmers to avoid long serial regions of execution (which would reduce total effective parallelism).

When using mutexes and condition variables, one tries to ensure that the usual case is to lock the mutex, access shared data, and unlock the mutex. Waiting on a condition variable should be a relatively rare situation. For example, when implementing a read-write lock, code that acquires a read-lock typically needs only to increment the count of readers (under mutual-exclusion) and return. The calling thread would actually wait on the condition variable only when there is already an active writer. So the efficiency of a synchronization operation is bounded by the cost of mutex lock/unlock and not by condition wait. Note that in the usual case there is no context switch.

This is not to say that the efficiency of condition waiting is unimportant. Since there needs to be at least one context switch per Ada rendezvous, the efficiency of waiting on a condition variable is important. The cost of waiting on a condition variable should be little more than the minimal cost for a context switch plus the time to unlock and lock the mutex.

Features of Mutexes and Condition Variables

It had been suggested that the mutex acquisition and release be decoupled from condition wait. This was rejected because it is the combined nature of the operation that, in fact, facilitates realtime implementations. Those implementations can atomically move a high-priority thread between the condition variable and the mutex in a manner that is transparent to the caller. This can prevent extra context switches and provide more deterministic acquisition of a mutex when the waiting thread is signaled. Thus, fairness and priority issues can be dealt with directly by the scheduling discipline. Furthermore, the current condition wait operation matches existing practice.

Scheduling Behavior of Mutexes and Condition Variables

Synchronization primitives that attempt to interfere with scheduling policy by specifying an ordering rule are considered undesirable. Threads waiting on mutexes and condition variables are selected to proceed in an order dependent upon the scheduling policy rather than in some fixed order (for example, FIFO or priority). Thus, the scheduling policy determines which thread(s) are awakened and allowed to proceed.

Timed Condition Wait

The *pthread_cond_timedwait()* function allows an application to give up waiting for a particular condition after a given amount of time. An example of its use follows:

```
(void) pthread_mutex_lock(&t.mn);
    t.waiters++;
    clock_gettime(CLOCK_REALTIME, &ts);
    ts.tv_sec += 5;
    rc = 0;
    while (! mypredicate(&t) && rc == 0)
        rc = pthread_cond_timedwait(&t.cond, &t.mn, &ts);
    t.waiters--;
    if (rc == 0) setmystate(&t);
(void) pthread_mutex_unlock(&t.mn);
```


32633 By making the timeout parameter absolute, it does not need to be recomputed each time the
32634 program checks its blocking predicate. If the timeout was relative, it would have to be
32635 recomputed before each call. This would be especially difficult since such code would need to
32636 take into account the possibility of extra wakeups that result from extra broadcasts or signals on
32637 the condition variable that occur before either the predicate is true or the timeout is due.

32638 FUTURE DIRECTIONS

32639 None.

32640 SEE ALSO

32641 *pthread_cond_signal()*, *pthread_cond_broadcast()*, the Base Definitions volume of
32642 IEEE Std 1003.1-2001, <pthread.h>

32643 CHANGE HISTORY

32644 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

32645 Issue 6

32646 The *pthread_cond_timedwait()* and *pthread_cond_wait()* functions are marked as part of the
32647 Threads option.

32648 The Open Group Corrigendum U021/9 is applied, correcting the prototype for the
32649 *pthread_cond_wait()* function.

32650 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding semantics for
32651 the Clock Selection option.

32652 The ERRORS section has an additional case for [EPERM] in response to IEEE PASC
32653 Interpretation 1003.1c #28.

32654 The **restrict** keyword is added to the *pthread_cond_timedwait()* and *pthread_cond_wait()*
32655 prototypes for alignment with the ISO/IEC 9899:1999 standard.

32656 **NAME**

32657 pthread_condattr_destroy, pthread_condattr_init — destroy and initialize the condition variable
 32658 attributes object

32659 **SYNOPSIS**

32660 THR `#include <pthread.h>`

32661 `int pthread_condattr_destroy(pthread_condattr_t *attr);`

32662 `int pthread_condattr_init(pthread_condattr_t *attr);`

32663

32664 **DESCRIPTION**

32665 The *pthread_condattr_destroy()* function shall destroy a condition variable attributes object; the
 32666 object becomes, in effect, uninitialized. An implementation may cause *pthread_condattr_destroy()*
 32667 to set the object referenced by *attr* to an invalid value. A destroyed *attr* attributes object can be
 32668 reinitialized using *pthread_condattr_init()*; the results of otherwise referencing the object after it
 32669 has been destroyed are undefined.

32670 The *pthread_condattr_init()* function shall initialize a condition variable attributes object *attr* with
 32671 the default value for all of the attributes defined by the implementation.

32672 Results are undefined if *pthread_condattr_init()* is called specifying an already initialized *attr*
 32673 attributes object.

32674 After a condition variable attributes object has been used to initialize one or more condition
 32675 variables, any function affecting the attributes object (including destruction) shall not affect any
 32676 previously initialized condition variables.

32677 This volume of IEEE Std 1003.1-2001 requires two attributes, the *clock* attribute and the *process-*
 32678 *shared* attribute.

32679 Additional attributes, their default values, and the names of the associated functions to get and
 32680 set those attribute values are implementation-defined.

32681 **RETURN VALUE**

32682 If successful, the *pthread_condattr_destroy()* and *pthread_condattr_init()* functions shall return
 32683 zero; otherwise, an error number shall be returned to indicate the error.

32684 **ERRORS**

32685 The *pthread_condattr_destroy()* function may fail if:

32686 [EINVAL] The value specified by *attr* is invalid.

32687 The *pthread_condattr_init()* function shall fail if:

32688 [ENOMEM] Insufficient memory exists to initialize the condition variable attributes object.

32689 These functions shall not return an error code of [EINTR].

32690 **EXAMPLES**

32691 None.

32692 **APPLICATION USAGE**

32693 None.

32694 **RATIONALE**

32695 See *pthread_attr_init()* and *pthread_mutex_init()*.

32696 A *process-shared* attribute has been defined for condition variables for the same reason it has been
 32697 defined for mutexes.

32698 FUTURE DIRECTIONS

32699 None.

32700 SEE ALSO

32701 *pthread_attr_destroy()*, *pthread_cond_destroy()*, *pthread_condattr_getpshared()*, *pthread_create()*,
32702 *pthread_mutex_destroy()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>

32703 CHANGE HISTORY

32704 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

32705 Issue 6

32706 The *pthread_condattr_destroy()* and *pthread_condattr_init()* functions are marked as part of the
32707 Threads option.

32708 **NAME**

32709 pthread_condattr_getclock, pthread_condattr_setclock — get and set the clock selection
 32710 condition variable attribute (**ADVANCED REALTIME**)

32711 **SYNOPSIS**

32712 THR CS #include <pthread.h>

```
32713 int pthread_condattr_getclock(const pthread_condattr_t *restrict attr,
32714 clockid_t *restrict clock_id);
32715 int pthread_condattr_setclock(pthread_condattr_t *attr,
32716 clockid_t clock_id);
32717
```

32718 **DESCRIPTION**

32719 The *pthread_condattr_getclock()* function shall obtain the value of the *clock* attribute from the
 32720 attributes object referenced by *attr*. The *pthread_condattr_setclock()* function shall set the *clock*
 32721 attribute in an initialized attributes object referenced by *attr*. If *pthread_condattr_setclock()* is
 32722 called with a *clock_id* argument that refers to a CPU-time clock, the call shall fail.

32723 The *clock* attribute is the clock ID of the clock that shall be used to measure the timeout service of
 32724 *pthread_cond_timedwait()*. The default value of the *clock* attribute shall refer to the system clock.

32725 **RETURN VALUE**

32726 If successful, the *pthread_condattr_getclock()* function shall return zero and store the value of the
 32727 clock attribute of *attr* into the object referenced by the *clock_id* argument. Otherwise, an error
 32728 number shall be returned to indicate the error.

32729 If successful, the *pthread_condattr_setclock()* function shall return zero; otherwise, an error
 32730 number shall be returned to indicate the error.

32731 **ERRORS**

32732 These functions may fail if:

32733 [EINVAL] The value specified by *attr* is invalid.

32734 The *pthread_condattr_setclock()* function may fail if:

32735 [EINVAL] The value specified by *clock_id* does not refer to a known clock, or is a CPU-
 32736 time clock.

32737 These functions shall not return an error code of [EINTR].

32738 **EXAMPLES**

32739 None.

32740 **APPLICATION USAGE**

32741 None.

32742 **RATIONALE**

32743 None.

32744 **FUTURE DIRECTIONS**

32745 None.

32746 **SEE ALSO**

32747 *pthread_cond_destroy()*, *pthread_cond_timedwait()*, *pthread_condattr_destroy()*,
 32748 *pthread_condattr_getpshared()* (on page 1041), *pthread_condattr_init()*,
 32749 *pthread_condattr_setpshared()* (on page 1045), *pthread_create()*, *pthread_mutex_init()*, the Base
 32750 Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

32751 **CHANGE HISTORY**

32752 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

32753 **NAME**

32754 pthread_condattr_getpshared, pthread_condattr_setpshared — get and set the process-shared
 32755 condition variable attributes

32756 **SYNOPSIS**

```
32757 THR TSH #include <pthread.h>

32758 int pthread_condattr_getpshared(const pthread_condattr_t *restrict attr,
32759 int *restrict pshared);
32760 int pthread_condattr_setpshared(pthread_condattr_t *attr,
32761 int pshared);
32762
```

32763 **DESCRIPTION**

32764 The *pthread_condattr_getpshared()* function shall obtain the value of the *process-shared* attribute
 32765 from the attributes object referenced by *attr*. The *pthread_condattr_setpshared()* function shall set
 32766 the *process-shared* attribute in an initialized attributes object referenced by *attr*.

32767 The *process-shared* attribute is set to PTHREAD_PROCESS_SHARED to permit a condition
 32768 variable to be operated upon by any thread that has access to the memory where the condition
 32769 variable is allocated, even if the condition variable is allocated in memory that is shared by
 32770 multiple processes. If the *process-shared* attribute is PTHREAD_PROCESS_PRIVATE, the
 32771 condition variable shall only be operated upon by threads created within the same process as the
 32772 thread that initialized the condition variable; if threads of differing processes attempt to operate
 32773 on such a condition variable, the behavior is undefined. The default value of the attribute is
 32774 PTHREAD_PROCESS_PRIVATE.

32775 **RETURN VALUE**

32776 If successful, the *pthread_condattr_setpshared()* function shall return zero; otherwise, an error
 32777 number shall be returned to indicate the error.

32778 If successful, the *pthread_condattr_getpshared()* function shall return zero and store the value of
 32779 the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise,
 32780 an error number shall be returned to indicate the error.

32781 **ERRORS**

32782 The *pthread_condattr_getpshared()* and *pthread_condattr_setpshared()* functions may fail if:

32783 [EINVAL] The value specified by *attr* is invalid.

32784 The *pthread_condattr_setpshared()* function may fail if:

32785 [EINVAL] The new value specified for the attribute is outside the range of legal values
 32786 for that attribute.

32787 These functions shall not return an error code of [EINTR].

32788 **EXAMPLES**

32789 None.

32790 **APPLICATION USAGE**

32791 None.

32792 **RATIONALE**

32793 None.

32794 FUTURE DIRECTIONS

32795 None.

32796 SEE ALSO

32797 *pthread_create()*, *pthread_cond_destroy()*, *pthread_condattr_destroy()*, *pthread_mutex_destroy()*, the
32798 Base Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>

32799 CHANGE HISTORY

32800 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

32801 Issue 6

32802 The *pthread_condattr_getpshared()* and *pthread_condattr_setpshared()* functions are marked as part
32803 of the Threads and Thread Process-Shared Synchronization options.

32804 The **restrict** keyword is added to the *pthread_condattr_getpshared()* prototype for alignment with
32805 the ISO/IEC 9899:1999 standard.

32806 **NAME**

32807 pthread_condattr_init — initialize the condition variable attributes object

32808 **SYNOPSIS**

32809 THR #include <pthread.h>

32810 int pthread_condattr_init(pthread_condattr_t *attr);

32811

32812 **DESCRIPTION**32813 Refer to *pthread_condattr_destroy()*.

32814 NAME

32815 pthread_condattr_setclock — set the clock selection condition variable attribute

32816 SYNOPSIS

32817 THR CS `#include <pthread.h>`

```
32818 int pthread_condattr_setclock(pthread_condattr_t *attr,  
32819 clockid_t clock_id);
```

32820

32821 DESCRIPTION

32822 Refer to *pthread_condattr_getclock()*.

32823 **NAME**

32824 pthread_condattr_setpshared — set the process-shared condition variable attribute

32825 **SYNOPSIS**32826 THR TSH `#include <pthread.h>`32827 `int pthread_condattr_setpshared(pthread_condattr_t *attr,`
32828 `int pshared);`

32829

32830 **DESCRIPTION**32831 Refer to *pthread_condattr_getpshared()*.

32832 NAME

32833 pthread_create — thread creation

32834 SYNOPSIS

32835 THR #include <pthread.h>

```

32836 int pthread_create(pthread_t *restrict thread,
32837     const pthread_attr_t *restrict attr,
32838     void *(*start_routine)(void*), void *restrict arg);
32839 
```

32840 DESCRIPTION

32841 The *pthread_create()* function shall create a new thread, with attributes specified by *attr*, within a
 32842 process. If *attr* is NULL, the default attributes shall be used. If the attributes specified by *attr* are
 32843 modified later, the thread's attributes shall not be affected. Upon successful completion,
 32844 *pthread_create()* shall store the ID of the created thread in the location referenced by *thread*.

32845 The thread is created executing *start_routine* with *arg* as its sole argument. If the *start_routine*
 32846 returns, the effect shall be as if there was an implicit call to *pthread_exit()* using the return value
 32847 of *start_routine* as the exit status. Note that the thread in which *main()* was originally invoked
 32848 differs from this. When it returns from *main()*, the effect shall be as if there was an implicit call
 32849 to *exit()* using the return value of *main()* as the exit status.

32850 The signal state of the new thread shall be initialized as follows:

- 32851 • The signal mask shall be inherited from the creating thread.
- 32852 • The set of signals pending for the new thread shall be empty.

32853 The floating-point environment shall be inherited from the creating thread.

32854 If *pthread_create()* fails, no new thread is created and the contents of the location referenced by
 32855 *thread* are undefined.

32856 TCT If `_POSIX_THREAD_CPUTIME` is defined, the new thread shall have a CPU-time clock
 32857 accessible, and the initial value of this clock shall be set to zero.

32858 RETURN VALUE

32859 If successful, the *pthread_create()* function shall return zero; otherwise, an error number shall be
 32860 returned to indicate the error.

32861 ERRORS

32862 The *pthread_create()* function shall fail if:

- 32863 [EAGAIN] The system lacked the necessary resources to create another thread, or the
 32864 system-imposed limit on the total number of threads in a process
 32865 {PTHREAD_THREADS_MAX} would be exceeded.
- 32866 [EINVAL] The value specified by *attr* is invalid.
- 32867 [EPERM] The caller does not have appropriate permission to set the required
 32868 scheduling parameters or scheduling policy.

32869 The *pthread_create()* function shall not return an error code of [EINTR].

32870 **EXAMPLES**

32871 None.

32872 **APPLICATION USAGE**

32873 None.

32874 **RATIONALE**

32875 A suggested alternative to *pthread_create()* would be to define two separate operations: create
 32876 and start. Some applications would find such behavior more natural. Ada, in particular,
 32877 separates the “creation” of a task from its “activation”.

32878 Splitting the operation was rejected by the standard developers for many reasons:

- 32879 • The number of calls required to start a thread would increase from one to two and thus place
 32880 an additional burden on applications that do not require the additional synchronization. The
 32881 second call, however, could be avoided by the additional complication of a start-up state
 32882 attribute.
- 32883 • An extra state would be introduced: “created but not started”. This would require the
 32884 standard to specify the behavior of the thread operations when the target has not yet started
 32885 executing.
- 32886 • For those applications that require such behavior, it is possible to simulate the two separate
 32887 steps with the facilities that are currently provided. The *start_routine()* can synchronize by
 32888 waiting on a condition variable that is signaled by the start operation.

32889 An Ada implementor can choose to create the thread at either of two points in the Ada program:
 32890 when the task object is created, or when the task is activated (generally at a “begin”). If the first
 32891 approach is adopted, the *start_routine()* needs to wait on a condition variable to receive the
 32892 order to begin “activation”. The second approach requires no such condition variable or extra
 32893 synchronization. In either approach, a separate Ada task control block would need to be created
 32894 when the task object is created to hold rendezvous queues, and so on.

32895 An extension of the preceding model would be to allow the state of the thread to be modified
 32896 between the create and start. This would allow the thread attributes object to be eliminated. This
 32897 has been rejected because:

- 32898 • All state in the thread attributes object has to be able to be set for the thread. This would
 32899 require the definition of functions to modify thread attributes. There would be no reduction
 32900 in the number of function calls required to set up the thread. In fact, for an application that
 32901 creates all threads using identical attributes, the number of function calls required to set up
 32902 the threads would be dramatically increased. Use of a thread attributes object permits the
 32903 application to make one set of attribute setting function calls. Otherwise, the set of attribute
 32904 setting function calls needs to be made for each thread creation.
- 32905 • Depending on the implementation architecture, functions to set thread state would require
 32906 kernel calls, or for other implementation reasons would not be able to be implemented as
 32907 macros, thereby increasing the cost of thread creation.
- 32908 • The ability for applications to segregate threads by class would be lost.

32909 Another suggested alternative uses a model similar to that for process creation, such as “thread
 32910 fork”. The fork semantics would provide more flexibility and the “create” function can be
 32911 implemented simply by doing a thread fork followed immediately by a call to the desired “start
 32912 routine” for the thread. This alternative has these problems:

- 32913 • For many implementations, the entire stack of the calling thread would need to be
 32914 duplicated, since in many architectures there is no way to determine the size of the calling
 32915 frame.

32916 • Efficiency is reduced since at least some part of the stack has to be copied, even though in
32917 most cases the thread never needs the copied context, since it merely calls the desired start
32918 routine.

32919 **FUTURE DIRECTIONS**

32920 None.

32921 **SEE ALSO**

32922 *fork()*, *pthread_exit()*, *pthread_join()*, the Base Definitions volume of IEEE Std 1003.1-2001,
32923 <pthread.h>

32924 **CHANGE HISTORY**

32925 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

32926 **Issue 6**

32927 The *pthread_create()* function is marked as part of the Threads option.

32928 The following new requirements on POSIX implementations derive from alignment with the
32929 Single UNIX Specification:

32930 • The [EPERM] mandatory error condition is added.

32931 The thread CPU-time clock semantics are added for alignment with IEEE Std 1003.1d-1999.

32932 The **restrict** keyword is added to the *pthread_create()* prototype for alignment with the
32933 ISO/IEC 9899:1999 standard.

32934 The DESCRIPTION is updated to make it explicit that the floating-point environment is
32935 inherited from the creating thread.

32936 **NAME**

32937 pthread_detach — detach a thread

32938 **SYNOPSIS**

32939 THR #include <pthread.h>

32940 int pthread_detach(pthread_t thread);

32941

32942 **DESCRIPTION**

32943 The *pthread_detach()* function shall indicate to the implementation that storage for the thread
 32944 *thread* can be reclaimed when that thread terminates. If *thread* has not terminated,
 32945 *pthread_detach()* shall not cause it to terminate. The effect of multiple *pthread_detach()* calls on
 32946 the same target thread is unspecified.

32947 **RETURN VALUE**

32948 If the call succeeds, *pthread_detach()* shall return 0; otherwise, an error number shall be returned
 32949 to indicate the error.

32950 **ERRORS**32951 The *pthread_detach()* function shall fail if:

32952	[EINVAL]	The implementation has detected that the value specified by <i>thread</i> does not refer to a joinable thread.
32953		
32954	[ESRCH]	No thread could be found corresponding to that specified by the given thread ID.
32955		

32956 The *pthread_detach()* function shall not return an error code of [EINTR].32957 **EXAMPLES**

32958 None.

32959 **APPLICATION USAGE**

32960 None.

32961 **RATIONALE**

32962 The *pthread_join()* or *pthread_detach()* functions should eventually be called for every thread that
 32963 is created so that storage associated with the thread may be reclaimed.

32964 It has been suggested that a “detach” function is not necessary; the *detachstate* thread creation
 32965 attribute is sufficient, since a thread need never be dynamically detached. However, need arises
 32966 in at least two cases:

- 32967 1. In a cancellation handler for a *pthread_join()* it is nearly essential to have a *pthread_detach()*
 32968 function in order to detach the thread on which *pthread_join()* was waiting. Without it, it
 32969 would be necessary to have the handler do another *pthread_join()* to attempt to detach the
 32970 thread, which would both delay the cancellation processing for an unbounded period and
 32971 introduce a new call to *pthread_join()*, which might itself need a cancellation handler. A
 32972 dynamic detach is nearly essential in this case.
- 32973 2. In order to detach the “initial thread” (as may be desirable in processes that set up server
 32974 threads).

32975 **FUTURE DIRECTIONS**

32976 None.

32977 SEE ALSO

32978 *pthread_join()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>

32979 CHANGE HISTORY

32980 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

32981 Issue 6

32982 The *pthread_detach()* function is marked as part of the Threads option.

32983 **NAME**

32984 pthread_equal — compare thread IDs

32985 **SYNOPSIS**

32986 THR #include <pthread.h>

32987 int pthread_equal(pthread_t t1, pthread_t t2);

32988

32989 **DESCRIPTION**32990 This function shall compare the thread IDs *t1* and *t2*.32991 **RETURN VALUE**32992 The *pthread_equal()* function shall return a non-zero value if *t1* and *t2* are equal; otherwise, zero
32993 shall be returned.32994 If either *t1* or *t2* are not valid thread IDs, the behavior is undefined.32995 **ERRORS**

32996 No errors are defined.

32997 The *pthread_equal()* function shall not return an error code of [EINTR].32998 **EXAMPLES**

32999 None.

33000 **APPLICATION USAGE**

33001 None.

33002 **RATIONALE**33003 Implementations may choose to define a thread ID as a structure. This allows additional
33004 flexibility and robustness over using an **int**. For example, a thread ID could include a sequence
33005 number that allows detection of “dangling IDs” (copies of a thread ID that has been detached).
33006 Since the C language does not support comparison on structure types, the *pthread_equal()*
33007 function is provided to compare thread IDs.33008 **FUTURE DIRECTIONS**

33009 None.

33010 **SEE ALSO**33011 *pthread_create()*, *pthread_self()*, the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>33012 **CHANGE HISTORY**

33013 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33014 **Issue 6**33015 The *pthread_equal()* function is marked as part of the Threads option.

33016 **NAME**

33017 pthread_exit — thread termination

33018 **SYNOPSIS**

33019 THR #include <pthread.h>

33020 void pthread_exit(void *value_ptr);

33021

33022 **DESCRIPTION**

33023 The *pthread_exit()* function shall terminate the calling thread and make the value *value_ptr*
33024 available to any successful join with the terminating thread. Any cancellation cleanup handlers
33025 that have been pushed and not yet popped shall be popped in the reverse order that they were
33026 pushed and then executed. After all cancellation cleanup handlers have been executed, if the
33027 thread has any thread-specific data, appropriate destructor functions shall be called in an
33028 unspecified order. Thread termination does not release any application visible process resources,
33029 including, but not limited to, mutexes and file descriptors, nor does it perform any process-level
33030 cleanup actions, including, but not limited to, calling any *atexit()* routines that may exist.

33031 An implicit call to *pthread_exit()* is made when a thread other than the thread in which *main()*
33032 was first invoked returns from the start routine that was used to create it. The function's return
33033 value shall serve as the thread's exit status.

33034 The behavior of *pthread_exit()* is undefined if called from a cancellation cleanup handler or
33035 destructor function that was invoked as a result of either an implicit or explicit call to
33036 *pthread_exit()*.

33037 After a thread has terminated, the result of access to local (auto) variables of the thread is
33038 undefined. Thus, references to local variables of the exiting thread should not be used for the
33039 *pthread_exit()* *value_ptr* parameter value.

33040 The process shall exit with an exit status of 0 after the last thread has been terminated. The
33041 behavior shall be as if the implementation called *exit()* with a zero argument at thread
33042 termination time.

33043 **RETURN VALUE**33044 The *pthread_exit()* function cannot return to its caller.33045 **ERRORS**

33046 No errors are defined.

33047 **EXAMPLES**

33048 None.

33049 **APPLICATION USAGE**

33050 None.

33051 **RATIONALE**

33052 The normal mechanism by which a thread terminates is to return from the routine that was
33053 specified in the *pthread_create()* call that started it. The *pthread_exit()* function provides the
33054 capability for a thread to terminate without requiring a return from the start routine of that
33055 thread, thereby providing a function analogous to *exit()*.

33056 Regardless of the method of thread termination, any cancellation cleanup handlers that have
33057 been pushed and not yet popped are executed, and the destructors for any existing thread-
33058 specific data are executed. This volume of IEEE Std 1003.1-2001 requires that cancellation
33059 cleanup handlers be popped and called in order. After all cancellation cleanup handlers have
33060 been executed, thread-specific data destructors are called, in an unspecified order, for each item
33061 of thread-specific data that exists in the thread. This ordering is necessary because cancellation

33062 cleanup handlers may rely on thread-specific data.

33063 As the meaning of the status is determined by the application (except when the thread has been
33064 canceled, in which case it is PTHREAD_CANCELED), the implementation has no idea what an
33065 illegal status value is, which is why no address error checking is done.

33066 **FUTURE DIRECTIONS**

33067 None.

33068 **SEE ALSO**

33069 *exit()*, *pthread_create()*, *pthread_join()*, the Base Definitions volume of IEEE Std 1003.1-2001,
33070 **<pthread.h>**

33071 **CHANGE HISTORY**

33072 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33073 **Issue 6**

33074 The *pthread_exit()* function is marked as part of the Threads option.

33075 **NAME**

33076 pthread_getconcurrency, pthread_setconcurrency — get and set the level of concurrency

33077 **SYNOPSIS**33078 XSI

```
#include <pthread.h>
```

33079

```
int pthread_getconcurrency(void);
```

33080

```
int pthread_setconcurrency(int new_level);
```

33081

33082 **DESCRIPTION**

33083 Unbound threads in a process may or may not be required to be simultaneously active. By
 33084 default, the threads implementation ensures that a sufficient number of threads are active so that
 33085 the process can continue to make progress. While this conserves system resources, it may not
 33086 produce the most effective level of concurrency.

33087 The *pthread_setconcurrency()* function allows an application to inform the threads
 33088 implementation of its desired concurrency level, *new_level*. The actual level of concurrency
 33089 provided by the implementation as a result of this function call is unspecified.

33090 If *new_level* is zero, it causes the implementation to maintain the concurrency level at its
 33091 discretion as if *pthread_setconcurrency()* had never been called.

33092 The *pthread_getconcurrency()* function shall return the value set by a previous call to the
 33093 *pthread_setconcurrency()* function. If the *pthread_setconcurrency()* function was not previously
 33094 called, this function shall return zero to indicate that the implementation is maintaining the
 33095 concurrency level.

33096 A call to *pthread_setconcurrency()* shall inform the implementation of its desired concurrency
 33097 level. The implementation shall use this as a hint, not a requirement.

33098 If an implementation does not support multiplexing of user threads on top of several kernel-
 33099 scheduled entities, the *pthread_setconcurrency()* and *pthread_getconcurrency()* functions are
 33100 provided for source code compatibility but they shall have no effect when called. To maintain
 33101 the function semantics, the *new_level* parameter is saved when *pthread_setconcurrency()* is called
 33102 so that a subsequent call to *pthread_getconcurrency()* shall return the same value.

33103 **RETURN VALUE**

33104 If successful, the *pthread_setconcurrency()* function shall return zero; otherwise, an error number
 33105 shall be returned to indicate the error.

33106 The *pthread_getconcurrency()* function shall always return the concurrency level set by a previous
 33107 call to *pthread_setconcurrency()*. If the *pthread_setconcurrency()* function has never been called,
 33108 *pthread_getconcurrency()* shall return zero.

33109 **ERRORS**

33110 The *pthread_setconcurrency()* function shall fail if:

33111 [EINVAL] The value specified by *new_level* is negative.

33112 [EAGAIN] The value specific by *new_level* would cause a system resource to be exceeded.

33113 These functions shall not return an error code of [EINTR].

33114 **EXAMPLES**

33115 None.

33116 **APPLICATION USAGE**

33117 Use of these functions changes the state of the underlying concurrency upon which the
33118 application depends. Library developers are advised to not use the *pthread_getconcurrency()* and
33119 *pthread_setconcurrency()* functions since their use may conflict with an applications use of these
33120 functions.

33121 **RATIONALE**

33122 None.

33123 **FUTURE DIRECTIONS**

33124 None.

33125 **SEE ALSO**

33126 The Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

33127 **CHANGE HISTORY**

33128 First released in Issue 5.

33129 **NAME**

33130 pthread_getcpuclockid — access a thread CPU-time clock (**ADVANCED REALTIME**
33131 **THREADS**)

33132 **SYNOPSIS**

33133 THR TCT #include <pthread.h>

33134 #include <time.h>

33135 int pthread_getcpuclockid(pthread_t thread_id, clockid_t *clock_id);

33136

33137 **DESCRIPTION**

33138 The *pthread_getcpuclockid()* function shall return in *clock_id* the clock ID of the CPU-time clock of
33139 the thread specified by *thread_id*, if the thread specified by *thread_id* exists.

33140 **RETURN VALUE**

33141 Upon successful completion, *pthread_getcpuclockid()* shall return zero; otherwise, an error
33142 number shall be returned to indicate the error.

33143 **ERRORS**

33144 The *pthread_getcpuclockid()* function may fail if:

33145 [ESRCH] The value specified by *thread_id* does not refer to an existing thread.

33146 **EXAMPLES**

33147 None.

33148 **APPLICATION USAGE**

33149 The *pthread_getcpuclockid()* function is part of the Thread CPU-Time Clocks option and need not
33150 be provided on all implementations.

33151 **RATIONALE**

33152 None.

33153 **FUTURE DIRECTIONS**

33154 None.

33155 **SEE ALSO**

33156 *clock_getcpuclockid()*, *clock_getres()*, *timer_create()*, the Base Definitions volume of
33157 IEEE Std 1003.1-2001, <pthread.h>, <time.h>

33158 **CHANGE HISTORY**

33159 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

33160 In the SYNOPSIS, the inclusion of <sys/types.h> is no longer required.

33161 NAME

33162 pthread_getschedparam, pthread_setschedparam — dynamic thread scheduling parameters
 33163 access (**REALTIME THREADS**)

33164 SYNOPSIS

33165 THR TPS #include <pthread.h>

```
33166 int pthread_getschedparam(pthread_t thread, int *restrict policy,
33167 struct sched_param *restrict param);
33168 int pthread_setschedparam(pthread_t thread, int policy,
33169 const struct sched_param *param);
33170
```

33171 DESCRIPTION

33172 The *pthread_getschedparam()* and *pthread_setschedparam()* functions shall, respectively, get and set
 33173 the scheduling policy and parameters of individual threads within a multi-threaded process to
 33174 be retrieved and set. For SCHED_FIFO and SCHED_RR, the only required member of the
 33175 **sched_param** structure is the priority *sched_priority*. For SCHED_OTHER, the affected
 33176 scheduling parameters are implementation-defined.

33177 The *pthread_getschedparam()* function shall retrieve the scheduling policy and scheduling
 33178 parameters for the thread whose thread ID is given by *thread* and shall store those values in
 33179 *policy* and *param*, respectively. The priority value returned from *pthread_getschedparam()* shall be
 33180 the value specified by the most recent *pthread_setschedparam()*, *pthread_setschedprio()*, or
 33181 *pthread_create()* call affecting the target thread. It shall not reflect any temporary adjustments to
 33182 its priority as a result of any priority inheritance or ceiling functions. The *pthread_setschedparam()*
 33183 function shall set the scheduling policy and associated scheduling parameters for the thread
 33184 whose thread ID is given by *thread* to the policy and associated parameters provided in *policy*
 33185 and *param*, respectively.

33186 The *policy* parameter may have the value SCHED_OTHER, SCHED_FIFO, or SCHED_RR. The
 33187 scheduling parameters for the SCHED_OTHER policy are implementation-defined. The
 33188 SCHED_FIFO and SCHED_RR policies shall have a single scheduling parameter, *priority*.

33189 TSP If _POSIX_THREAD_SPORADIC_SERVER is defined, then the *policy* argument may have the
 33190 value SCHED_SPORADIC, with the exception for the *pthread_setschedparam()* function that if the
 33191 scheduling policy was not SCHED_SPORADIC at the time of the call, it is implementation-
 33192 defined whether the function is supported; in other words, the implementation need not allow
 33193 the application to dynamically change the scheduling policy to SCHED_SPORADIC. The
 33194 sporadic server scheduling policy has the associated parameters *sched_ss_low_priority*,
 33195 *sched_ss_repl_period*, *sched_ss_init_budget*, *sched_priority*, and *sched_ss_max_repl*. The specified
 33196 *sched_ss_repl_period* shall be greater than or equal to the specified *sched_ss_init_budget* for the
 33197 function to succeed; if it is not, then the function shall fail. The value of *sched_ss_max_repl* shall
 33198 be within the inclusive range [1,{SS_REPL_MAX}] for the function to succeed; if not, the function
 33199 shall fail.

33200 If the *pthread_setschedparam()* function fails, the scheduling parameters shall not be changed for
 33201 the target thread.

33202 RETURN VALUE

33203 If successful, the *pthread_getschedparam()* and *pthread_setschedparam()* functions shall return zero;
 33204 otherwise, an error number shall be returned to indicate the error.

33205 **ERRORS**

33206 The *pthread_getschedparam()* function may fail if:

33207 [ESRCH] The value specified by *thread* does not refer to an existing thread.

33208 The *pthread_setschedparam()* function may fail if:

33209 [EINVAL] The value specified by *policy* or one of the scheduling parameters associated
33210 with the scheduling policy *policy* is invalid.

33211 [ENOTSUP] An attempt was made to set the policy or scheduling parameters to an
33212 unsupported value.

33213 TSP [ENOTSUP] An attempt was made to dynamically change the scheduling policy to
33214 SCHED_SPORADIC, and the implementation does not support this change.

33215 [EPERM] The caller does not have the appropriate permission to set either the
33216 scheduling parameters or the scheduling policy of the specified thread.

33217 [EPERM] The implementation does not allow the application to modify one of the
33218 parameters to the value specified.

33219 [ESRCH] The value specified by *thread* does not refer to a existing thread.

33220 These functions shall not return an error code of [EINTR].

33221 **EXAMPLES**

33222 None.

33223 **APPLICATION USAGE**

33224 None.

33225 **RATIONALE**

33226 None.

33227 **FUTURE DIRECTIONS**

33228 None.

33229 **SEE ALSO**

33230 *pthread_setschedprio()*, *sched_getparam()*, *sched_getscheduler()*, the Base Definitions volume of
33231 IEEE Std 1003.1-2001, <pthread.h>, <sched.h>

33232 **CHANGE HISTORY**

33233 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33234 **Issue 6**

33235 The *pthread_getschedparam()* and *pthread_setschedparam()* functions are marked as part of the
33236 Threads and Thread Execution Scheduling options.

33237 The [ENOSYS] error condition has been removed as stubs need not be provided if an
33238 implementation does not support the Thread Execution Scheduling option.

33239 The Open Group Corrigendum U026/2 is applied, correcting the prototype for the
33240 *pthread_setschedparam()* function so that its second argument is of type **int**.

33241 The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

33242 The **restrict** keyword is added to the *pthread_getschedparam()* prototype for alignment with the
33243 ISO/IEC 9899:1999 standard.

33244 The Open Group Corrigendum U047/1 is applied.

33245 IEEE PASC Interpretation 1003.1 #96 is applied, noting that priority values can also be set by a
33246 call to the *pthread_setschedprio()* function.

33247 **NAME**

33248 pthread_getspecific, pthread_setspecific — thread-specific data management

33249 **SYNOPSIS**33250 THR `#include <pthread.h>`33251 `void *pthread_getspecific(pthread_key_t key);`33252 `int pthread_setspecific(pthread_key_t key, const void *value);`

33253

33254 **DESCRIPTION**33255 The *pthread_getspecific()* function shall return the value currently bound to the specified *key* on
33256 behalf of the calling thread.33257 The *pthread_setspecific()* function shall associate a thread-specific *value* with a *key* obtained via a
33258 previous call to *pthread_key_create()*. Different threads may bind different values to the same
33259 key. These values are typically pointers to blocks of dynamically allocated memory that have
33260 been reserved for use by the calling thread.33261 The effect of calling *pthread_getspecific()* or *pthread_setspecific()* with a *key* value not obtained
33262 from *pthread_key_create()* or after *key* has been deleted with *pthread_key_delete()* is undefined.33263 Both *pthread_getspecific()* and *pthread_setspecific()* may be called from a thread-specific data
33264 destructor function. A call to *pthread_getspecific()* for the thread-specific data key being
33265 destroyed shall return the value NULL, unless the value is changed (after the destructor starts)
33266 by a call to *pthread_setspecific()*. Calling *pthread_setspecific()* from a thread-specific data
33267 destructor routine may result either in lost storage (after at least
33268 PTHREAD_DESTRUCTOR_ITERATIONS attempts at destruction) or in an infinite loop.

33269 Both functions may be implemented as macros.

33270 **RETURN VALUE**33271 The *pthread_getspecific()* function shall return the thread-specific data value associated with the
33272 given *key*. If no thread-specific data value is associated with *key*, then the value NULL shall be
33273 returned.33274 If successful, the *pthread_setspecific()* function shall return zero; otherwise, an error number shall
33275 be returned to indicate the error.33276 **ERRORS**33277 No errors are returned from *pthread_getspecific()*.33278 The *pthread_setspecific()* function shall fail if:

33279 [ENOMEM] Insufficient memory exists to associate the value with the key.

33280 The *pthread_setspecific()* function may fail if:

33281 [EINVAL] The key value is invalid.

33282 These functions shall not return an error code of [EINTR].

33283 **EXAMPLES**

33284 None.

33285 **APPLICATION USAGE**

33286 None.

33287 **RATIONALE**

33288 Performance and ease-of-use of *pthread_getspecific()* are critical for functions that rely on
33289 maintaining state in thread-specific data. Since no errors are required to be detected by it, and
33290 since the only error that could be detected is the use of an invalid key, the function to
33291 *pthread_getspecific()* has been designed to favor speed and simplicity over error reporting.

33292 **FUTURE DIRECTIONS**

33293 None.

33294 **SEE ALSO**33295 *pthread_key_create()*, the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>33296 **CHANGE HISTORY**

33297 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33298 **Issue 6**

33299 The *pthread_getspecific()* and *pthread_setspecific()* functions are marked as part of the Threads
33300 option.

33301 IEEE PASC Interpretation 1003.1c #3 (Part 6) is applied, updating the DESCRIPTION.

33302 NAME

33303 pthread_join — wait for thread termination

33304 SYNOPSIS

33305 THR #include <pthread.h>

33306 int pthread_join(pthread_t thread, void **value_ptr);

33307

33308 DESCRIPTION

33309 The *pthread_join()* function shall suspend execution of the calling thread until the target *thread*
 33310 terminates, unless the target *thread* has already terminated. On return from a successful
 33311 *pthread_join()* call with a non-NULL *value_ptr* argument, the value passed to *pthread_exit()* by
 33312 the terminating thread shall be made available in the location referenced by *value_ptr*. When a
 33313 *pthread_join()* returns successfully, the target thread has been terminated. The results of multiple
 33314 simultaneous calls to *pthread_join()* specifying the same target thread are undefined. If the
 33315 thread calling *pthread_join()* is canceled, then the target thread shall not be detached.

33316 It is unspecified whether a thread that has exited but remains unjoined counts against
 33317 {PTHREAD_THREADS_MAX}.

33318 RETURN VALUE

33319 If successful, the *pthread_join()* function shall return zero; otherwise, an error number shall be
 33320 returned to indicate the error.

33321 ERRORS

33322 The *pthread_join()* function shall fail if:

33323 [EINVAL] The implementation has detected that the value specified by *thread* does not
 33324 refer to a joinable thread.

33325 [ESRCH] No thread could be found corresponding to that specified by the given thread
 33326 ID.

33327 The *pthread_join()* function may fail if:

33328 [EDEADLK] A deadlock was detected or the value of *thread* specifies the calling thread.

33329 The *pthread_join()* function shall not return an error code of [EINTR].

33330 EXAMPLES

33331 An example of thread creation and deletion follows:

```

33332 typedef struct {
33333     int *ar;
33334     long n;
33335 } subarray;

33336 void *
33337 incer(void *arg)
33338 {
33339     long i;

33340     for (i = 0; i < ((subarray *)arg)->n; i++)
33341         ((subarray *)arg)->ar[i]++;
33342 }

33343 int main(void)
33344 {
33345     int         ar[1000000];

```



```

33346     pthread_t  th1, th2;
33347     subarray   sb1, sb2;

33348     sb1.ar = &ar[0];
33349     sb1.n  = 500000;
33350     (void) pthread_create(&th1, NULL, incer, &sb1);

33351     sb2.ar = &ar[500000];
33352     sb2.n  = 500000;
33353     (void) pthread_create(&th2, NULL, incer, &sb2);

33354     (void) pthread_join(th1, NULL);
33355     (void) pthread_join(th2, NULL);
33356     return 0;
33357 }
```

33358 APPLICATION USAGE

33359 None.

33360 RATIONALE

33361 The *pthread_join()* function is a convenience that has proven useful in multi-threaded
 33362 applications. It is true that a programmer could simulate this function if it were not provided by
 33363 passing extra state as part of the argument to the *start_routine()*. The terminating thread would
 33364 set a flag to indicate termination and broadcast a condition that is part of that state; a joining
 33365 thread would wait on that condition variable. While such a technique would allow a thread to
 33366 wait on more complex conditions (for example, waiting for multiple threads to terminate),
 33367 waiting on individual thread termination is considered widely useful. Also, including the
 33368 *pthread_join()* function in no way precludes a programmer from coding such complex waits.
 33369 Thus, while not a primitive, including *pthread_join()* in this volume of IEEE Std 1003.1-2001 was
 33370 considered valuable.

33371 The *pthread_join()* function provides a simple mechanism allowing an application to wait for a
 33372 thread to terminate. After the thread terminates, the application may then choose to clean up
 33373 resources that were used by the thread. For instance, after *pthread_join()* returns, any
 33374 application-provided stack storage could be reclaimed.

33375 The *pthread_join()* or *pthread_detach()* function should eventually be called for every thread that
 33376 is created with the *detachstate* attribute set to *PTHREAD_CREATE_JOINABLE* so that storage
 33377 associated with the thread may be reclaimed.

33378 The interaction between *pthread_join()* and cancelation is well-defined for the following reasons:

- 33379 • The *pthread_join()* function, like all other non-async-cancel-safe functions, can only be called
- 33380 with deferred cancelability type.
- 33381 • Cancelation cannot occur in the disabled cancelability state.

33382 Thus, only the default cancelability state need be considered. As specified, either the
 33383 *pthread_join()* call is canceled, or it succeeds, but not both. The difference is obvious to the
 33384 application, since either a cancelation handler is run or *pthread_join()* returns. There are no race
 33385 conditions since *pthread_join()* was called in the deferred cancelability state.

33386 FUTURE DIRECTIONS

33387 None.

33388 SEE ALSO

33389 *pthread_create()*, *wait()*, the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

33390 CHANGE HISTORY

33391 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33392 Issue 6

33393 The *pthread_join()* function is marked as part of the Threads option.

33394 **NAME**

33395 pthread_key_create — thread-specific data key creation

33396 **SYNOPSIS**

33397 THR #include <pthread.h>

33398 int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));

33399

33400 **DESCRIPTION**

33401 The *pthread_key_create()* function shall create a thread-specific data key visible to all threads in
 33402 the process. Key values provided by *pthread_key_create()* are opaque objects used to locate
 33403 thread-specific data. Although the same key value may be used by different threads, the values
 33404 bound to the key by *pthread_setspecific()* are maintained on a per-thread basis and persist for the
 33405 life of the calling thread.

33406 Upon key creation, the value NULL shall be associated with the new key in all active threads.
 33407 Upon thread creation, the value NULL shall be associated with all defined keys in the new
 33408 thread.

33409 An optional destructor function may be associated with each key value. At thread exit, if a key
 33410 value has a non-NULL destructor pointer, and the thread has a non-NULL value associated with
 33411 that key, the value of the key is set to NULL, and then the function pointed to is called with the
 33412 previously associated value as its sole argument. The order of destructor calls is unspecified if
 33413 more than one destructor exists for a thread when it exits.

33414 If, after all the destructors have been called for all non-NULL values with associated destructors,
 33415 there are still some non-NULL values with associated destructors, then the process is repeated.
 33416 If, after at least {PTHREAD_DESTRUCTOR_ITERATIONS} iterations of destructor calls for
 33417 outstanding non-NULL values, there are still some non-NULL values with associated
 33418 destructors, implementations may stop calling destructors, or they may continue calling
 33419 destructors until no non-NULL values with associated destructors exist, even though this might
 33420 result in an infinite loop.

33421 **RETURN VALUE**

33422 If successful, the *pthread_key_create()* function shall store the newly created key value at *key and
 33423 shall return zero. Otherwise, an error number shall be returned to indicate the error.

33424 **ERRORS**33425 The *pthread_key_create()* function shall fail if:

33426 [EAGAIN] The system lacked the necessary resources to create another thread-specific
 33427 data key, or the system-imposed limit on the total number of keys per process
 33428 {PTHREAD_KEYS_MAX} has been exceeded.

33429 [ENOMEM] Insufficient memory exists to create the key.

33430 The *pthread_key_create()* function shall not return an error code of [EINTR].

33431 **EXAMPLES**

33432 The following example demonstrates a function that initializes a thread-specific data key when
 33433 it is first called, and associates a thread-specific object with each calling thread, initializing this
 33434 object when necessary.

```

33435 static pthread_key_t key;
33436 static pthread_once_t key_once = PTHREAD_ONCE_INIT;

33437 static void
33438 make_key()
33439 {
33440     (void) pthread_key_create(&key, NULL);
33441 }

33442 func()
33443 {
33444     void *ptr;

33445     (void) pthread_once(&key_once, make_key);
33446     if ((ptr = pthread_getspecific(key)) == NULL) {
33447         ptr = malloc(OBJECT_SIZE);
33448         ...
33449         (void) pthread_setspecific(key, ptr);
33450     }
33451     ...
33452 }
```

33453 Note that the key has to be initialized before *pthread_getspecific()* or *pthread_setspecific()* can be
 33454 used. The *pthread_key_create()* call could either be explicitly made in a module initialization
 33455 routine, or it can be done implicitly by the first call to a module as in this example. Any attempt
 33456 to use the key before it is initialized is a programming error, making the code below incorrect.

```

33457 static pthread_key_t key;

33458 func()
33459 {
33460     void *ptr;

33461     /* KEY NOT INITIALIZED!!! THIS WON'T WORK!!! */
33462     if ((ptr = pthread_getspecific(key)) == NULL &&
33463         pthread_setspecific(key, NULL) != 0) {
33464         pthread_key_create(&key, NULL);
33465         ...
33466     }
33467 }
```

33468 **APPLICATION USAGE**

33469 None.

33470 RATIONALE

33471 **Destructor Functions**

33472 Normally, the value bound to a key on behalf of a particular thread is a pointer to storage
33473 allocated dynamically on behalf of the calling thread. The destructor functions specified with
33474 *pthread_key_create()* are intended to be used to free this storage when the thread exits. Thread
33475 cancellation cleanup handlers cannot be used for this purpose because thread-specific data may
33476 persist outside the lexical scope in which the cancellation cleanup handlers operate.

33477 If the value associated with a key needs to be updated during the lifetime of the thread, it may
33478 be necessary to release the storage associated with the old value before the new value is bound.
33479 Although the *pthread_setspecific()* function could do this automatically, this feature is not needed
33480 often enough to justify the added complexity. Instead, the programmer is responsible for freeing
33481 the stale storage:

```
33482 pthread_getspecific(key, &old);  
33483 new = allocate();  
33484 destructor(old);  
33485 pthread_setspecific(key, new);
```

33486 **Note:** The above example could leak storage if run with asynchronous cancellation enabled. No such
33487 problems occur in the default cancellation state if no cancellation points occur between the get
33488 and set.

33489 There is no notion of a destructor-safe function. If an application does not call *pthread_exit()*
33490 from a signal handler, or if it blocks any signal whose handler may call *pthread_exit()* while
33491 calling async-unsafe functions, all functions may be safely called from destructors.

33492 **Non-Idempotent Data Key Creation**

33493 There were requests to make *pthread_key_create()* idempotent with respect to a given *key* address
33494 parameter. This would allow applications to call *pthread_key_create()* multiple times for a given
33495 *key* address and be guaranteed that only one key would be created. Doing so would require the
33496 key value to be previously initialized (possibly at compile time) to a known null value and
33497 would require that implicit mutual-exclusion be performed based on the address and contents of
33498 the *key* parameter in order to guarantee that exactly one key would be created.

33499 Unfortunately, the implicit mutual-exclusion would not be limited to only *pthread_key_create()*.
33500 On many implementations, implicit mutual-exclusion would also have to be performed by
33501 *pthread_getspecific()* and *pthread_setspecific()* in order to guard against using incompletely stored
33502 or not-yet-visible key values. This could significantly increase the cost of important operations,
33503 particularly *pthread_getspecific()*.

33504 Thus, this proposal was rejected. The *pthread_key_create()* function performs no implicit
33505 synchronization. It is the responsibility of the programmer to ensure that it is called exactly once
33506 per key before use of the key. Several straightforward mechanisms can already be used to
33507 accomplish this, including calling explicit module initialization functions, using mutexes, and
33508 using *pthread_once()*. This places no significant burden on the programmer, introduces no
33509 possibly confusing *ad hoc* implicit synchronization mechanism, and potentially allows
33510 commonly used thread-specific data operations to be more efficient.

33511 **FUTURE DIRECTIONS**

33512 None.

33513 SEE ALSO

33514 *pthread_getspecific()*, *pthread_key_delete()*, the Base Definitions volume of IEEE Std 1003.1-2001,
33515 **<pthread.h>**

33516 CHANGE HISTORY

33517 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33518 Issue 6

33519 The *pthread_key_create()* function is marked as part of the Threads option.

33520 IEEE PASC Interpretation 1003.1c #8 is applied, updating the DESCRIPTION.

33521 **NAME**

33522 pthread_key_delete — thread-specific data key deletion

33523 **SYNOPSIS**

33524 THR #include <pthread.h>

33525 int pthread_key_delete(pthread_key_t key);

33526

33527 **DESCRIPTION**

33528 The *pthread_key_delete()* function shall delete a thread-specific data key previously returned by
 33529 *pthread_key_create()*. The thread-specific data values associated with *key* need not be NULL at
 33530 the time *pthread_key_delete()* is called. It is the responsibility of the application to free any
 33531 application storage or perform any cleanup actions for data structures related to the deleted key
 33532 or associated thread-specific data in any threads; this cleanup can be done either before or after
 33533 *pthread_key_delete()* is called. Any attempt to use *key* following the call to *pthread_key_delete()*
 33534 results in undefined behavior.

33535 The *pthread_key_delete()* function shall be callable from within destructor functions. No
 33536 destructor functions shall be invoked by *pthread_key_delete()*. Any destructor function that may
 33537 have been associated with *key* shall no longer be called upon thread exit.

33538 **RETURN VALUE**

33539 If successful, the *pthread_key_delete()* function shall return zero; otherwise, an error number shall
 33540 be returned to indicate the error.

33541 **ERRORS**33542 The *pthread_key_delete()* function may fail if:33543 [EINVAL] The *key* value is invalid.33544 The *pthread_key_delete()* function shall not return an error code of [EINTR].33545 **EXAMPLES**

33546 None.

33547 **APPLICATION USAGE**

33548 None.

33549 **RATIONALE**

33550 A thread-specific data key deletion function has been included in order to allow the resources
 33551 associated with an unused thread-specific data key to be freed. Unused thread-specific data keys
 33552 can arise, among other scenarios, when a dynamically loaded module that allocated a key is
 33553 unloaded.

33554 Conforming applications are responsible for performing any cleanup actions needed for data
 33555 structures associated with the key to be deleted, including data referenced by thread-specific
 33556 data values. No such cleanup is done by *pthread_key_delete()*. In particular, destructor functions
 33557 are not called. There are several reasons for this division of responsibility:

- 33558 1. The associated destructor functions used to free thread-specific data at thread exit time are
 33559 only guaranteed to work correctly when called in the thread that allocated the thread-
 33560 specific data. (Destructors themselves may utilize thread-specific data.) Thus, they cannot
 33561 be used to free thread-specific data in other threads at key deletion time. Attempting to
 33562 have them called by other threads at key deletion time would require other threads to be
 33563 asynchronously interrupted. But since interrupted threads could be in an arbitrary state,
 33564 including holding locks necessary for the destructor to run, this approach would fail. In
 33565 general, there is no safe mechanism whereby an implementation could free thread-specific
 33566 data at key deletion time.

33567 2. Even if there were a means of safely freeing thread-specific data associated with keys to be
33568 deleted, doing so would require that implementations be able to enumerate the threads
33569 with non-NULL data and potentially keep them from creating more thread-specific data
33570 while the key deletion is occurring. This special case could cause extra synchronization in
33571 the normal case, which would otherwise be unnecessary.

33572 For an application to know that it is safe to delete a key, it has to know that all the threads that
33573 might potentially ever use the key do not attempt to use it again. For example, it could know this
33574 if all the client threads have called a cleanup procedure declaring that they are through with the
33575 module that is being shut down, perhaps by setting a reference count to zero.

33576 **FUTURE DIRECTIONS**

33577 None.

33578 **SEE ALSO**

33579 *pthread_key_create()*, the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

33580 **CHANGE HISTORY**

33581 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33582 **Issue 6**

33583 The *pthread_key_delete()* function is marked as part of the Threads option.

33584 **NAME**

33585 pthread_kill — send a signal to a thread

33586 **SYNOPSIS**

33587 THR #include <signal.h>

33588 int pthread_kill(pthread_t thread, int sig);

33589

33590 **DESCRIPTION**33591 The *pthread_kill()* function shall request that a signal be delivered to the specified thread.33592 As in *kill()*, if *sig* is zero, error checking shall be performed but no signal shall actually be sent.33593 **RETURN VALUE**33594 Upon successful completion, the function shall return a value of zero. Otherwise, the function
33595 shall return an error number. If the *pthread_kill()* function fails, no signal shall be sent.33596 **ERRORS**33597 The *pthread_kill()* function shall fail if:33598 [ESRCH] No thread could be found corresponding to that specified by the given thread
33599 ID.33600 [EINVAL] The value of the *sig* argument is an invalid or unsupported signal number.33601 The *pthread_kill()* function shall not return an error code of [EINTR].33602 **EXAMPLES**

33603 None.

33604 **APPLICATION USAGE**33605 The *pthread_kill()* function provides a mechanism for asynchronously directing a signal at a
33606 thread in the calling process. This could be used, for example, by one thread to affect broadcast
33607 delivery of a signal to a set of threads.33608 Note that *pthread_kill()* only causes the signal to be handled in the context of the given thread;
33609 the signal action (termination or stopping) affects the process as a whole.33610 **RATIONALE**

33611 None.

33612 **FUTURE DIRECTIONS**

33613 None.

33614 **SEE ALSO**33615 *kill()*, *pthread_self()*, *raise()*, the Base Definitions volume of IEEE Std 1003.1-2001, <signal.h>33616 **CHANGE HISTORY**

33617 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33618 **Issue 6**33619 The *pthread_kill()* function is marked as part of the Threads option.

33620 The APPLICATION USAGE section is added.

33621 NAME

33622 pthread_mutex_destroy, pthread_mutex_init — destroy and initialize a mutex

33623 SYNOPSIS

33624 THR #include <pthread.h>

```

33625 int pthread_mutex_destroy(pthread_mutex_t *mutex);
33626 int pthread_mutex_init(pthread_mutex_t *restrict mutex,
33627     const pthread_mutexattr_t *restrict attr);
33628 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
33629

```

33630 DESCRIPTION

33631 The *pthread_mutex_destroy()* function shall destroy the mutex object referenced by *mutex*; the
 33632 mutex object becomes, in effect, uninitialized. An implementation may cause
 33633 *pthread_mutex_destroy()* to set the object referenced by *mutex* to an invalid value. A destroyed
 33634 mutex object can be reinitialized using *pthread_mutex_init()*; the results of otherwise referencing
 33635 the object after it has been destroyed are undefined.

33636 It shall be safe to destroy an initialized mutex that is unlocked. Attempting to destroy a locked
 33637 mutex results in undefined behavior.

33638 The *pthread_mutex_init()* function shall initialize the mutex referenced by *mutex* with attributes
 33639 specified by *attr*. If *attr* is NULL, the default mutex attributes are used; the effect shall be the
 33640 same as passing the address of a default mutex attributes object. Upon successful initialization,
 33641 the state of the mutex becomes initialized and unlocked.

33642 Only *mutex* itself may be used for performing synchronization. The result of referring to copies
 33643 of *mutex* in calls to *pthread_mutex_lock()*, *pthread_mutex_trylock()*, *pthread_mutex_unlock()*, and
 33644 *pthread_mutex_destroy()* is undefined.

33645 Attempting to initialize an already initialized mutex results in undefined behavior.

33646 In cases where default mutex attributes are appropriate, the macro
 33647 PTHREAD_MUTEX_INITIALIZER can be used to initialize mutexes that are statically allocated.
 33648 The effect shall be equivalent to dynamic initialization by a call to *pthread_mutex_init()* with
 33649 parameter *attr* specified as NULL, except that no error checks are performed.

33650 RETURN VALUE

33651 If successful, the *pthread_mutex_destroy()* and *pthread_mutex_init()* functions shall return zero;
 33652 otherwise, an error number shall be returned to indicate the error.

33653 The [EBUSY] and [EINVAL] error checks, if implemented, act as if they were performed
 33654 immediately at the beginning of processing for the function and shall cause an error return prior
 33655 to modifying the state of the mutex specified by *mutex*.

33656 ERRORS

33657 The *pthread_mutex_destroy()* function may fail if:

33658 [EBUSY] The implementation has detected an attempt to destroy the object referenced
 33659 by *mutex* while it is locked or referenced (for example, while being used in a
 33660 *pthread_cond_timedwait()* or *pthread_cond_wait()*) by another thread.

33661 [EINVAL] The value specified by *mutex* is invalid.

33662 The *pthread_mutex_init()* function shall fail if:

33663 [EAGAIN] The system lacked the necessary resources (other than memory) to initialize
 33664 another mutex.

- 33665 [ENOMEM] Insufficient memory exists to initialize the mutex.
- 33666 [EPERM] The caller does not have the privilege to perform the operation.
- 33667 The *pthread_mutex_init()* function may fail if:
- 33668 [EBUSY] The implementation has detected an attempt to reinitialize the object
33669 referenced by *mutex*, a previously initialized, but not yet destroyed, mutex.
- 33670 [EINVAL] The value specified by *attr* is invalid.
- 33671 These functions shall not return an error code of [EINTR].

33672 **EXAMPLES**

33673 None.

33674 **APPLICATION USAGE**

33675 None.

33676 **RATIONALE**33677 **Alternate Implementations Possible**

33678 This volume of IEEE Std 1003.1-2001 supports several alternative implementations of mutexes.
33679 An implementation may store the lock directly in the object of type **pthread_mutex_t**.
33680 Alternatively, an implementation may store the lock in the heap and merely store a pointer,
33681 handle, or unique ID in the mutex object. Either implementation has advantages or may be
33682 required on certain hardware configurations. So that portable code can be written that is
33683 invariant to this choice, this volume of IEEE Std 1003.1-2001 does not define assignment or
33684 equality for this type, and it uses the term “initialize” to reinforce the (more restrictive) notion
33685 that the lock may actually reside in the mutex object itself.

33686 Note that this precludes an over-specification of the type of the mutex or condition variable and
33687 motivates the opaqueness of the type.

33688 An implementation is permitted, but not required, to have *pthread_mutex_destroy()* store an
33689 illegal value into the mutex. This may help detect erroneous programs that try to lock (or
33690 otherwise reference) a mutex that has already been destroyed.

33691 **Tradeoff Between Error Checks and Performance Supported**

33692 Many of the error checks were made optional in order to let implementations trade off
33693 performance *versus* degree of error checking according to the needs of their specific applications
33694 and execution environment. As a general rule, errors or conditions caused by the system (such as
33695 insufficient memory) always need to be reported, but errors due to an erroneously coded
33696 application (such as failing to provide adequate synchronization to prevent a mutex from being
33697 deleted while in use) are made optional.

33698 A wide range of implementations is thus made possible. For example, an implementation
33699 intended for application debugging may implement all of the error checks, but an
33700 implementation running a single, provably correct application under very tight performance
33701 constraints in an embedded computer might implement minimal checks. An implementation
33702 might even be provided in two versions, similar to the options that compilers provide: a full-
33703 checking, but slower version; and a limited-checking, but faster version. To forbid this
33704 optionality would be a disservice to users.

33705 By carefully limiting the use of “undefined behavior” only to things that an erroneous (badly
33706 coded) application might do, and by defining that resource-not-available errors are mandatory,
33707 this volume of IEEE Std 1003.1-2001 ensures that a fully-conforming application is portable

across the full range of implementations, while not forcing all implementations to add overhead to check for numerous things that a correct program never does.

Why No Limits are Defined

Defining symbols for the maximum number of mutexes and condition variables was considered but rejected because the number of these objects may change dynamically. Furthermore, many implementations place these objects into application memory; thus, there is no explicit maximum.

Static Initializers for Mutexes and Condition Variables

Providing for static initialization of statically allocated synchronization objects allows modules with private static synchronization variables to avoid runtime initialization tests and overhead. Furthermore, it simplifies the coding of self-initializing modules. Such modules are common in C libraries, where for various reasons the design calls for self-initialization instead of requiring an explicit module initialization function to be called. An example use of static initialization follows.

Without static initialization, a self-initializing routine *foo()* might look as follows:

```
static pthread_once_t foo_once = PTHREAD_ONCE_INIT;
static pthread_mutex_t foo_mutex;

void foo_init()
{
    pthread_mutex_init(&foo_mutex, NULL);
}

void foo()
{
    pthread_once(&foo_once, foo_init);
    pthread_mutex_lock(&foo_mutex);
    /* Do work. */
    pthread_mutex_unlock(&foo_mutex);
}
```

With static initialization, the same routine could be coded as follows:

```
static pthread_mutex_t foo_mutex = PTHREAD_MUTEX_INITIALIZER;

void foo()
{
    pthread_mutex_lock(&foo_mutex);
    /* Do work. */
    pthread_mutex_unlock(&foo_mutex);
}
```

Note that the static initialization both eliminates the need for the initialization test inside *pthread_once()* and the fetch of *&foo_mutex* to learn the address to be passed to *pthread_mutex_lock()* or *pthread_mutex_unlock()*.

Thus, the C code written to initialize static objects is simpler on all systems and is also faster on a large class of systems; those where the (entire) synchronization object can be stored in application memory.

Yet the locking performance question is likely to be raised for machines that require mutexes to be allocated out of special memory. Such machines actually have to have mutexes and possibly

condition variables contain pointers to the actual hardware locks. For static initialization to work on such machines, *pthread_mutex_lock()* also has to test whether or not the pointer to the actual lock has been allocated. If it has not, *pthread_mutex_lock()* has to initialize it before use. The reservation of such resources can be made when the program is loaded, and hence return codes have not been added to mutex locking and condition variable waiting to indicate failure to complete initialization.

This runtime test in *pthread_mutex_lock()* would at first seem to be extra work; an extra test is required to see whether the pointer has been initialized. On most machines this would actually be implemented as a fetch of the pointer, testing the pointer against zero, and then using the pointer if it has already been initialized. While the test might seem to add extra work, the extra effort of testing a register is usually negligible since no extra memory references are actually done. As more and more machines provide caches, the real expenses are memory references, not instructions executed.

Alternatively, depending on the machine architecture, there are often ways to eliminate *all* overhead in the most important case: on the lock operations that occur *after* the lock has been initialized. This can be done by shifting more overhead to the less frequent operation: initialization. Since out-of-line mutex allocation also means that an address has to be dereferenced to find the actual lock, one technique that is widely applicable is to have static initialization store a bogus value for that address; in particular, an address that causes a machine fault to occur. When such a fault occurs upon the first attempt to lock such a mutex, validity checks can be done, and then the correct address for the actual lock can be filled in. Subsequent lock operations incur no extra overhead since they do not “fault”. This is merely one technique that can be used to support static initialization, while not adversely affecting the performance of lock acquisition. No doubt there are other techniques that are highly machine-dependent.

The locking overhead for machines doing out-of-line mutex allocation is thus similar for modules being implicitly initialized, where it is improved for those doing mutex allocation entirely inline. The inline case is thus made much faster, and the out-of-line case is not significantly worse.

Besides the issue of locking performance for such machines, a concern is raised that it is possible that threads would serialize contending for initialization locks when attempting to finish initializing statically allocated mutexes. (Such finishing would typically involve taking an internal lock, allocating a structure, storing a pointer to the structure in the mutex, and releasing the internal lock.) First, many implementations would reduce such serialization by hashing on the mutex address. Second, such serialization can only occur a bounded number of times. In particular, it can happen at most as many times as there are statically allocated synchronization objects. Dynamically allocated objects would still be initialized via *pthread_mutex_init()* or *pthread_cond_init()*.

Finally, if none of the above optimization techniques for out-of-line allocation yields sufficient performance for an application on some implementation, the application can avoid static initialization altogether by explicitly initializing all synchronization objects with the corresponding *pthread_*_init()* functions, which are supported by all implementations. An implementation can also document the tradeoffs and advise which initialization technique is more efficient for that particular implementation.

Destroying Mutexes

A mutex can be destroyed immediately after it is unlocked. For example, consider the following code:

```
struct obj {
pthread_mutex_t om;
    int refcnt;
    ...
};

obj_done(struct obj *op)
{
    pthread_mutex_lock(&op->om);
    if (--op->refcnt == 0) {
        pthread_mutex_unlock(&op->om);
        (A)    pthread_mutex_destroy(&op->om);
        (B)    free(op);
    } else
        (C)    pthread_mutex_unlock(&op->om);
}
```

In this case *obj* is reference counted and *obj_done()* is called whenever a reference to the object is dropped. Implementations are required to allow an object to be destroyed and freed and potentially unmapped (for example, lines A and B) immediately after the object is unlocked (line C).

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_mutex_getprioceiling(), *pthread_mutex_lock()*, *pthread_mutex_timedlock()*, *pthread_mutexattr_getpshared()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_mutex_destroy()* and *pthread_mutex_init()* functions are marked as part of the Threads option.

The *pthread_mutex_timedlock()* function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

IEEE PASC Interpretation 1003.1c #34 is applied, updating the DESCRIPTION.

The **restrict** keyword is added to the *pthread_mutex_init()* prototype for alignment with the ISO/IEC 9899:1999 standard.

33833 **NAME**

33834 pthread_mutex_getprioceiling, pthread_mutex_setprioceiling — get and set the priority ceiling
 33835 of a mutex (**REALTIME THREADS**)

33836 **SYNOPSIS**

33837 THR TPP #include <pthread.h>

```
33838 int pthread_mutex_getprioceiling(const pthread_mutex_t *restrict mutex,
33839 int *restrict prioceiling);
33840 int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mutex,
33841 int prioceiling, int *restrict old_ceiling);
33842
```

33843 **DESCRIPTION**

33844 The *pthread_mutex_getprioceiling()* function shall return the current priority ceiling of the mutex.

33845 The *pthread_mutex_setprioceiling()* function shall either lock the mutex if it is unlocked, or block
 33846 until it can successfully lock the mutex, then it shall change the mutex's priority ceiling and
 33847 release the mutex. When the change is successful, the previous value of the priority ceiling shall
 33848 be returned in *old_ceiling*. The process of locking the mutex need not adhere to the priority
 33849 protect protocol.

33850 If the *pthread_mutex_setprioceiling()* function fails, the mutex priority ceiling shall not be
 33851 changed.

33852 **RETURN VALUE**

33853 If successful, the *pthread_mutex_getprioceiling()* and *pthread_mutex_setprioceiling()* functions shall
 33854 return zero; otherwise, an error number shall be returned to indicate the error.

33855 **ERRORS**

33856 The *pthread_mutex_getprioceiling()* and *pthread_mutex_setprioceiling()* functions may fail if:

33857 [EINVAL] The priority requested by *prioceiling* is out of range.
 33858 [EINVAL] The value specified by *mutex* does not refer to a currently existing mutex.
 33859 [EPERM] The caller does not have the privilege to perform the operation.
 33860 These functions shall not return an error code of [EINTR].

33861 **EXAMPLES**

33862 None.

33863 **APPLICATION USAGE**

33864 None.

33865 **RATIONALE**

33866 None.

33867 **FUTURE DIRECTIONS**

33868 None.

33869 **SEE ALSO**

33870 *pthread_mutex_destroy()*, *pthread_mutex_lock()*, *pthread_mutex_timedlock()*, the Base Definitions
 33871 volume of IEEE Std 1003.1-2001, <pthread.h>

33872 **CHANGE HISTORY**

33873 First released in Issue 5. Included for alignment with the POSIX Threads Extension.
 33874 Marked as part of the Realtime Threads Feature Group.

33875 **Issue 6**

33876 The *pthread_mutex_getprioceiling()* and *pthread_mutex_setprioceiling()* functions are marked as
33877 part of the Threads and Thread Priority Protection options.

33878 The [ENOSYS] error condition has been removed as stubs need not be provided if an
33879 implementation does not support the Thread Priority Protection option.

33880 The [ENOSYS] error denoting non-support of the priority ceiling protocol for mutexes has been
33881 removed. This is because if the implementation provides the functions (regardless of whether
33882 `_POSIX_PTHREAD_PRIO_PROTECT` is defined), they must function as in the DESCRIPTION
33883 and therefore the priority ceiling protocol for mutexes is supported.

33884 The *pthread_mutex_timedlock()* function is added to the SEE ALSO section for alignment with
33885 IEEE Std 1003.1d-1999.

33886 The **restrict** keyword is added to the *pthread_mutex_getprioceiling()* and
33887 *pthread_mutex_setprioceiling()* prototypes for alignment with the ISO/IEC 9899:1999 standard.

33888 **NAME**

33889 pthread_mutex_init — initialize a mutex

33890 **SYNOPSIS**

33891 THR #include <pthread.h>

33892 int pthread_mutex_init(pthread_mutex_t *restrict mutex,

33893 const pthread_mutexattr_t *restrict attr);

33894 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

33895

33896 **DESCRIPTION**33897 Refer to *pthread_mutex_destroy()*.

33898 NAME

33899 pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock — lock and unlock a
33900 mutex

33901 SYNOPSIS

33902 THR `#include <pthread.h>`

```
33903 int pthread_mutex_lock(pthread_mutex_t *mutex);
33904 int pthread_mutex_trylock(pthread_mutex_t *mutex);
33905 int pthread_mutex_unlock(pthread_mutex_t *mutex);
33906
```

33907 DESCRIPTION

33908 The mutex object referenced by *mutex* shall be locked by calling *pthread_mutex_lock()*. If the
33909 mutex is already locked, the calling thread shall block until the mutex becomes available. This
33910 operation shall return with the mutex object referenced by *mutex* in the locked state with the
33911 calling thread as its owner.

33912 XSI If the mutex type is PTHREAD_MUTEX_NORMAL, deadlock detection shall not be provided.
33913 Attempting to relock the mutex causes deadlock. If a thread attempts to unlock a mutex that it
33914 has not locked or a mutex which is unlocked, undefined behavior results.

33915 If the mutex type is PTHREAD_MUTEX_ERRORCHECK, then error checking shall be provided.
33916 If a thread attempts to relock a mutex that it has already locked, an error shall be returned. If a
33917 thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error
33918 shall be returned.

33919 If the mutex type is PTHREAD_MUTEX_RECURSIVE, then the mutex shall maintain the
33920 concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock
33921 count shall be set to one. Every time a thread relocks this mutex, the lock count shall be
33922 incremented by one. Each time the thread unlocks the mutex, the lock count shall be
33923 decremented by one. When the lock count reaches zero, the mutex shall become available for
33924 other threads to acquire. If a thread attempts to unlock a mutex that it has not locked or a mutex
33925 which is unlocked, an error shall be returned.

33926 If the mutex type is PTHREAD_MUTEX_DEFAULT, attempting to recursively lock the mutex
33927 results in undefined behavior. Attempting to unlock the mutex if it was not locked by the calling
33928 thread results in undefined behavior. Attempting to unlock the mutex if it is not locked results in
33929 undefined behavior.

33930 The *pthread_mutex_trylock()* function shall be equivalent to *pthread_mutex_lock()*, except that if
33931 the mutex object referenced by *mutex* is currently locked (by any thread, including the current
33932 thread), the call shall return immediately. If the mutex type is PTHREAD_MUTEX_RECURSIVE
33933 and the mutex is currently owned by the calling thread, the mutex lock count shall be
33934 incremented by one and the *pthread_mutex_trylock()* function shall immediately return success.

33935 XSI The *pthread_mutex_unlock()* function shall release the mutex object referenced by *mutex*. The
33936 manner in which a mutex is released is dependent upon the mutex's type attribute. If there are
33937 threads blocked on the mutex object referenced by *mutex* when *pthread_mutex_unlock()* is called,
33938 resulting in the mutex becoming available, the scheduling policy shall determine which thread
33939 shall acquire the mutex.

33940 XSI (In the case of PTHREAD_MUTEX_RECURSIVE mutexes, the mutex shall become available
33941 when the count reaches zero and the calling thread no longer has any locks on this mutex.)

33942 If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the
33943 thread shall resume waiting for the mutex as if it was not interrupted.

33944 **RETURN VALUE**

33945 If successful, the *pthread_mutex_lock()* and *pthread_mutex_unlock()* functions shall return zero;
 33946 otherwise, an error number shall be returned to indicate the error.

33947 The *pthread_mutex_trylock()* function shall return zero if a lock on the mutex object referenced by
 33948 *mutex* is acquired. Otherwise, an error number is returned to indicate the error.

33949 **ERRORS**

33950 The *pthread_mutex_lock()* and *pthread_mutex_trylock()* functions shall fail if:

33951 [EINVAL] The *mutex* was created with the protocol attribute having the value
 33952 PTHREAD_PRIO_PROTECT and the calling thread's priority is higher than
 33953 the mutex's current priority ceiling.

33954 The *pthread_mutex_trylock()* function shall fail if:

33955 [EBUSY] The *mutex* could not be acquired because it was already locked.

33956 The *pthread_mutex_lock()*, *pthread_mutex_trylock()*, and *pthread_mutex_unlock()* functions may
 33957 fail if:

33958 [EINVAL] The value specified by *mutex* does not refer to an initialized mutex object.

33959 XSI [EAGAIN] The mutex could not be acquired because the maximum number of recursive
 33960 locks for *mutex* has been exceeded.

33961 The *pthread_mutex_lock()* function may fail if:

33962 [EDEADLK] The current thread already owns the mutex.

33963 The *pthread_mutex_unlock()* function may fail if:

33964 [EPERM] The current thread does not own the mutex.

33965 These functions shall not return an error code of [EINTR].

33966 **EXAMPLES**

33967 None.

33968 **APPLICATION USAGE**

33969 None.

33970 **RATIONALE**

33971 Mutex objects are intended to serve as a low-level primitive from which other thread
 33972 synchronization functions can be built. As such, the implementation of mutexes should be as
 33973 efficient as possible, and this has ramifications on the features available at the interface.

33974 The mutex functions and the particular default settings of the mutex attributes have been
 33975 motivated by the desire to not preclude fast, inlined implementations of mutex locking and
 33976 unlocking.

33977 For example, deadlocking on a double-lock is explicitly allowed behavior in order to avoid
 33978 requiring more overhead in the basic mechanism than is absolutely necessary. (More “friendly”
 33979 mutexes that detect deadlock or that allow multiple locking by the same thread are easily
 33980 constructed by the user via the other mechanisms provided. For example, *pthread_self()* can be
 33981 used to record mutex ownership.) Implementations might also choose to provide such extended
 33982 features as options via special mutex attributes.

33983 Since most attributes only need to be checked when a thread is going to be blocked, the use of
 33984 attributes does not slow the (common) mutex-locking case.

33985 Likewise, while being able to extract the thread ID of the owner of a mutex might be desirable, it
33986 would require storing the current thread ID when each mutex is locked, and this could incur
33987 unacceptable levels of overhead. Similar arguments apply to a *mutex_tryunlock* operation.

33988 FUTURE DIRECTIONS

33989 None.

33990 SEE ALSO

33991 *pthread_mutex_destroy()*, *pthread_mutex_timedlock()*, the Base Definitions volume of
33992 IEEE Std 1003.1-2001, <pthread.h>

33993 CHANGE HISTORY

33994 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33995 Issue 6

33996 The *pthread_mutex_lock()*, *pthread_mutex_trylock()*, and *pthread_mutex_unlock()* functions are
33997 marked as part of the Threads option.

33998 The following new requirements on POSIX implementations derive from alignment with the
33999 Single UNIX Specification:

- 34000 • The behavior when attempting to relock a mutex is defined.

34001 The *pthread_mutex_timedlock()* function is added to the SEE ALSO section for alignment with
34002 IEEE Std 1003.1d-1999.

34003 **NAME**

34004 pthread_mutex_setprioceiling — change the priority ceiling of a mutex (**REALTIME**
34005 **THREADS**)

34006 **SYNOPSIS**

34007 THR TPP #include <pthread.h>

34008 int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mutex,
34009 int prioceiling, int *restrict old_ceiling);

34010

34011 **DESCRIPTION**

34012 Refer to *pthread_mutex_getprioceiling()*.

34013 NAME

34014 pthread_mutex_timedlock — lock a mutex (ADVANCED REALTIME)

34015 SYNOPSIS

34016 THR TMO #include <pthread.h>

34017 #include <time.h>

34018 int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,

34019 const struct timespec *restrict abs_timeout);

34020

34021 DESCRIPTION

34022 The *pthread_mutex_timedlock()* function shall lock the mutex object referenced by *mutex*. If the
 34023 mutex is already locked, the calling thread shall block until the mutex becomes available as in
 34024 the *pthread_mutex_lock()* function. If the mutex cannot be locked without waiting for another
 34025 thread to unlock the mutex, this wait shall be terminated when the specified timeout expires.

34026 The timeout shall expire when the absolute time specified by *abs_timeout* passes, as measured by
 34027 the clock on which timeouts are based (that is, when the value of that clock equals or exceeds
 34028 *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time
 34029 of the call.

34030 TMR If the Timers option is supported, the timeout shall be based on the CLOCK_REALTIME clock; if
 34031 the Timers option is not supported, the timeout shall be based on the system clock as returned
 34032 by the *time()* function.

34033 The resolution of the timeout shall be the resolution of the clock on which it is based. The
 34034 **timespec** data type is defined in the <time.h> header.

34035 Under no circumstance shall the function fail with a timeout if the mutex can be locked
 34036 immediately. The validity of the *abs_timeout* parameter need not be checked if the mutex can be
 34037 locked immediately.

34038 As a consequence of the priority inheritance rules (for mutexes initialized with the
 34039 PRIO_INHERIT protocol), if a timed mutex wait is terminated because its timeout expires, the
 34040 priority of the owner of the mutex shall be adjusted as necessary to reflect the fact that this
 34041 thread is no longer among the threads waiting for the mutex.

34042 RETURN VALUE

34043 If successful, the *pthread_mutex_timedlock()* function shall return zero; otherwise, an error
 34044 number shall be returned to indicate the error.

34045 ERRORS

34046 The *pthread_mutex_timedlock()* function shall fail if:

34047 [EINVAL] The mutex was created with the protocol attribute having the value
 34048 PTHREAD_PRIO_PROTECT and the calling thread's priority is higher than
 34049 the mutex' current priority ceiling.

34050 [EINVAL] The process or thread would have blocked, and the *abs_timeout* parameter
 34051 specified a nanoseconds field value less than zero or greater than or equal to
 34052 1 000 million.

34053 [ETIMEDOUT] The mutex could not be locked before the specified timeout expired.

34054 The *pthread_mutex_timedlock()* function may fail if:

34055 [EINVAL] The value specified by *mutex* does not refer to an initialized mutex object.

34056 XSI [EAGAIN] The mutex could not be acquired because the maximum number of recursive
34057 locks for *mutex* has been exceeded.

34058 [EDEADLK] The current thread already owns the mutex.

34059 This function shall not return an error code of [EINTR].

34060 EXAMPLES

34061 None.

34062 APPLICATION USAGE

34063 The *pthread_mutex_timedlock()* function is part of the Threads and Timeouts options and need
34064 not be provided on all implementations.

34065 RATIONALE

34066 None.

34067 FUTURE DIRECTIONS

34068 None.

34069 SEE ALSO

34070 *pthread_mutex_destroy()*, *pthread_mutex_lock()*, *pthread_mutex_trylock()*, *time()*, the Base
34071 Definitions volume of IEEE Std 1003.1-2001, <pthread.h>, <time.h>

34072 CHANGE HISTORY

34073 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

34074 NAME

34075 pthread_mutex_trylock, pthread_mutex_unlock — lock and unlock a mutex

34076 SYNOPSIS

34077 THR `#include <pthread.h>`

34078 `int pthread_mutex_trylock(pthread_mutex_t *mutex);`

34079 `int pthread_mutex_unlock(pthread_mutex_t *mutex);`

34080

34081 DESCRIPTION

34082 Refer to *pthread_mutex_lock()*.

34083 **NAME**

34084 pthread_mutexattr_destroy, pthread_mutexattr_init — destroy and initialize the mutex
 34085 attributes object

34086 **SYNOPSIS**

34087 THR #include <pthread.h>

34088 int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);

34089 int pthread_mutexattr_init(pthread_mutexattr_t *attr);

34090

34091 **DESCRIPTION**

34092 The *pthread_mutexattr_destroy()* function shall destroy a mutex attributes object; the object
 34093 becomes, in effect, uninitialized. An implementation may cause *pthread_mutexattr_destroy()* to
 34094 set the object referenced by *attr* to an invalid value. A destroyed *attr* attributes object can be
 34095 reinitialized using *pthread_mutexattr_init()*; the results of otherwise referencing the object after it
 34096 has been destroyed are undefined.

34097 The *pthread_mutexattr_init()* function shall initialize a mutex attributes object *attr* with the
 34098 default value for all of the attributes defined by the implementation.

34099 Results are undefined if *pthread_mutexattr_init()* is called specifying an already initialized *attr*
 34100 attributes object.

34101 After a mutex attributes object has been used to initialize one or more mutexes, any function
 34102 affecting the attributes object (including destruction) shall not affect any previously initialized
 34103 mutexes.

34104 **RETURN VALUE**

34105 Upon successful completion, *pthread_mutexattr_destroy()* and *pthread_mutexattr_init()* shall
 34106 return zero; otherwise, an error number shall be returned to indicate the error.

34107 **ERRORS**

34108 The *pthread_mutexattr_destroy()* function may fail if:

34109 [EINVAL] The value specified by *attr* is invalid.

34110 The *pthread_mutexattr_init()* function shall fail if:

34111 [ENOMEM] Insufficient memory exists to initialize the mutex attributes object.

34112 These functions shall not return an error code of [EINTR].

34113 **EXAMPLES**

34114 None.

34115 **APPLICATION USAGE**

34116 None.

34117 **RATIONALE**

34118 See *pthread_attr_init()* for a general explanation of attributes. Attributes objects allow
 34119 implementations to experiment with useful extensions and permit extension of this volume of
 34120 IEEE Std 1003.1-2001 without changing the existing functions. Thus, they provide for future
 34121 extensibility of this volume of IEEE Std 1003.1-2001 and reduce the temptation to standardize
 34122 prematurely on semantics that are not yet widely implemented or understood.

34123 Examples of possible additional mutex attributes that have been discussed are *spin_only*,
 34124 *limited_spin*, *no_spin*, *recursive*, and *metered*. (To explain what the latter attributes might mean:
 34125 recursive mutexes would allow for multiple re-locking by the current owner; metered mutexes
 34126 would transparently keep records of queue length, wait time, and so on.) Since there is not yet

wide agreement on the usefulness of these resulting from shared implementation and usage experience, they are not yet specified in this volume of IEEE Std 1003.1-2001. Mutex attributes objects, however, make it possible to test out these concepts for possible standardization at a later time.

Mutex Attributes and Performance

Care has been taken to ensure that the default values of the mutex attributes have been defined such that mutexes initialized with the defaults have simple enough semantics so that the locking and unlocking can be done with the equivalent of a test-and-set instruction (plus possibly a few other basic instructions).

There is at least one implementation method that can be used to reduce the cost of testing at lock-time if a mutex has non-default attributes. One such method that an implementation can employ (and this can be made fully transparent to fully conforming POSIX applications) is to secretly pre-lock any mutexes that are initialized to non-default attributes. Any later attempt to lock such a mutex causes the implementation to branch to the “slow path” as if the mutex were unavailable; then, on the slow path, the implementation can do the “real work” to lock a non-default mutex. The underlying unlock operation is more complicated since the implementation never really wants to release the pre-lock on this kind of mutex. This illustrates that, depending on the hardware, there may be certain optimizations that can be used so that whatever mutex attributes are considered “most frequently used” can be processed most efficiently.

Process Shared Memory and Synchronization

The existence of memory mapping functions in this volume of IEEE Std 1003.1-2001 leads to the possibility that an application may allocate the synchronization objects from this section in memory that is accessed by multiple processes (and therefore, by threads of multiple processes).

In order to permit such usage, while at the same time keeping the usual case (that is, usage within a single process) efficient, a *process-shared* option has been defined.

If an implementation supports the `_POSIX_THREAD_PROCESS_SHARED` option, then the *process-shared* attribute can be used to indicate that mutexes or condition variables may be accessed by threads of multiple processes.

The default setting of `PTHREAD_PROCESS_PRIVATE` has been chosen for the *process-shared* attribute so that the most efficient forms of these synchronization objects are created by default.

Synchronization variables that are initialized with the `PTHREAD_PROCESS_PRIVATE` *process-shared* attribute may only be operated on by threads in the process that initialized them. Synchronization variables that are initialized with the `PTHREAD_PROCESS_SHARED` *process-shared* attribute may be operated on by any thread in any process that has access to it. In particular, these processes may exist beyond the lifetime of the initializing process. For example, the following code implements a simple counting semaphore in a mapped file that may be used by many processes.

```
/* sem.h */
struct semaphore {
    pthread_mutex_t lock;
    pthread_cond_t nonzero;
    unsigned count;
};
typedef struct semaphore semaphore_t;

semaphore_t *semaphore_create(char *semaphore_name);
semaphore_t *semaphore_open(char *semaphore_name);
```



```

34173 void semaphore_post(semaphore_t *semap);
34174 void semaphore_wait(semaphore_t *semap);
34175 void semaphore_close(semaphore_t *semap);

34176 /* sem.c */
34177 #include <sys/types.h>
34178 #include <sys/stat.h>
34179 #include <sys/mman.h>
34180 #include <fcntl.h>
34181 #include <pthread.h>
34182 #include "sem.h"

34183 semaphore_t *
34184 semaphore_create(char *semaphore_name)
34185 {
34186     int fd;
34187     semaphore_t *semap;
34188     pthread_mutexattr_t psharedm;
34189     pthread_condattr_t psharedc;

34190     fd = open(semaphore_name, O_RDWR | O_CREAT | O_EXCL, 0666);
34191     if (fd < 0)
34192         return (NULL);
34193     (void) ftruncate(fd, sizeof(semaphore_t));
34194     (void) pthread_mutexattr_init(&psharedm);
34195     (void) pthread_mutexattr_setpshared(&psharedm,
34196         PTHREAD_PROCESS_SHARED);
34197     (void) pthread_condattr_init(&psharedc);
34198     (void) pthread_condattr_setpshared(&psharedc,
34199         PTHREAD_PROCESS_SHARED);
34200     semap = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
34201         PROT_READ | PROT_WRITE, MAP_SHARED,
34202         fd, 0);
34203     close (fd);
34204     (void) pthread_mutex_init(&semap->lock, &psharedm);
34205     (void) pthread_cond_init(&semap->nonzero, &psharedc);
34206     semap->count = 0;
34207     return (semap);
34208 }

34209 semaphore_t *
34210 semaphore_open(char *semaphore_name)
34211 {
34212     int fd;
34213     semaphore_t *semap;

34214     fd = open(semaphore_name, O_RDWR, 0666);
34215     if (fd < 0)
34216         return (NULL);
34217     semap = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
34218         PROT_READ | PROT_WRITE, MAP_SHARED,
34219         fd, 0);
34220     close (fd);
34221     return (semap);
34222 }

```



```

34223 void
34224 semaphore_post(semaphore_t *semap)
34225 {
34226     pthread_mutex_lock(&semap->lock);
34227     if (semap->count == 0)
34228         pthread_cond_signal(&semap->nonzero);
34229     semap->count++;
34230     pthread_mutex_unlock(&semap->lock);
34231 }
34232
34233 void
34234 semaphore_wait(semaphore_t *semap)
34235 {
34236     pthread_mutex_lock(&semap->lock);
34237     while (semap->count == 0)
34238         pthread_cond_wait(&semap->nonzero, &semap->lock);
34239     semap->count--;
34240     pthread_mutex_unlock(&semap->lock);
34241 }

```

```

34241 void
34242 semaphore_close(semaphore_t *semap)
34243 {
34244     munmap((void *) semap, sizeof(semaphore_t));
34245 }

```

34246 The following code is for three separate processes that create, post, and wait on a semaphore in
34247 the file **/tmp/semaphore**. Once the file is created, the post and wait programs increment and
34248 decrement the counting semaphore (waiting and waking as required) even though they did not
34249 initialize the semaphore.

```

34250 /* create.c */
34251 #include "pthread.h"
34252 #include "sem.h"
34253
34254 int
34255 main()
34256 {
34257     semaphore_t *semap;
34258
34259     semap = semaphore_create("/tmp/semaphore");
34260     if (semap == NULL)
34261         exit(1);
34262     semaphore_close(semap);
34263     return (0);
34264 }
34265
34266 /* post */
34267 #include "pthread.h"
34268 #include "sem.h"
34269
34270 int
34271 main()
34272 {
34273     semaphore_t *semap;

```



```

34270         semap = semaphore_open("/tmp/semaphore");
34271         if (semap == NULL)
34272             exit(1);
34273         semaphore_post(semap);
34274         semaphore_close(semap);
34275         return (0);
34276     }

34277     /* wait */
34278     #include "pthread.h"
34279     #include "sem.h"

34280     int
34281     main()
34282     {
34283         semaphore_t *semap;

34284         semap = semaphore_open("/tmp/semaphore");
34285         if (semap == NULL)
34286             exit(1);
34287         semaphore_wait(semap);
34288         semaphore_close(semap);
34289         return (0);
34290     }

```

34291 FUTURE DIRECTIONS

34292 None.

34293 SEE ALSO

34294 *pthread_cond_destroy()*, *pthread_create()*, *pthread_mutex_destroy()*, *pthread_mutexattr_destroy()*, the
34295 Base Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>

34296 CHANGE HISTORY

34297 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34298 Issue 6

34299 The *pthread_mutexattr_destroy()* and *pthread_mutexattr_init()* functions are marked as part of the
34300 Threads option.

34301 IEEE PASC Interpretation 1003.1c #27 is applied, updating the ERRORS section.

34302 NAME

34303 pthread_mutexattr_getprioceiling, pthread_mutexattr_setprioceiling — get and set the
 34304 prioceiling attribute of the mutex attributes object (**REALTIME THREADS**)

34305 SYNOPSIS

34306 THR TPP #include <pthread.h>

```
34307 int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *
34308     restrict attr, int *restrict prioceiling);
34309 int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
34310     int prioceiling);
34311
```

34312 DESCRIPTION

34313 The *pthread_mutexattr_getprioceiling()* and *pthread_mutexattr_setprioceiling()* functions,
 34314 respectively, shall get and set the priority ceiling attribute of a mutex attributes object pointed to
 34315 by *attr* which was previously created by the function *pthread_mutexattr_init()*.

34316 The *prioceiling* attribute contains the priority ceiling of initialized mutexes. The values of
 34317 *prioceiling* are within the maximum range of priorities defined by SCHED_FIFO.

34318 The *prioceiling* attribute defines the priority ceiling of initialized mutexes, which is the minimum
 34319 priority level at which the critical section guarded by the mutex is executed. In order to avoid
 34320 priority inversion, the priority ceiling of the mutex shall be set to a priority higher than or equal
 34321 to the highest priority of all the threads that may lock that mutex. The values of *prioceiling* are
 34322 within the maximum range of priorities defined under the SCHED_FIFO scheduling policy.

34323 RETURN VALUE

34324 Upon successful completion, the *pthread_mutexattr_getprioceiling()* and
 34325 *pthread_mutexattr_setprioceiling()* functions shall return zero; otherwise, an error number shall be
 34326 returned to indicate the error.

34327 ERRORS

34328 The *pthread_mutexattr_getprioceiling()* and *pthread_mutexattr_setprioceiling()* functions may fail if:

34329 [EINVAL] The value specified by *attr* or *prioceiling* is invalid.

34330 [EPERM] The caller does not have the privilege to perform the operation.

34331 These functions shall not return an error code of [EINTR].

34332 EXAMPLES

34333 None.

34334 APPLICATION USAGE

34335 None.

34336 RATIONALE

34337 None.

34338 FUTURE DIRECTIONS

34339 None.

34340 SEE ALSO

34341 *pthread_cond_destroy()*, *pthread_create()*, *pthread_mutex_destroy()*, the Base Definitions volume of
 34342 IEEE Std 1003.1-2001, <pthread.h>

34343 **CHANGE HISTORY**

34344 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34345 Marked as part of the Realtime Threads Feature Group.

34346 **Issue 6**

34347 The *pthread_mutexattr_getprioceiling()* and *pthread_mutexattr_setprioceiling()* functions are
34348 marked as part of the Threads and Thread Priority Protection options.

34349 The [ENOSYS] error condition has been removed as stubs need not be provided if an
34350 implementation does not support the Thread Priority Protection option.

34351 The [ENOTSUP] error condition has been removed since these functions do not have a *protocol*
34352 argument.

34353 The **restrict** keyword is added to the *pthread_mutexattr_getprioceiling()* prototype for alignment
34354 with the ISO/IEC 9899:1999 standard.

34355 NAME

34356 pthread_mutexattr_getprotocol, pthread_mutexattr_setprotocol — get and set the protocol
 34357 attribute of the mutex attributes object (**REALTIME THREADS**)

34358 SYNOPSIS

34359 THR #include <pthread.h>

34360 TPP|TPI int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *
 34361 restrict attr, int *restrict protocol);
 34362 int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
 34363 int protocol);
 34364

34365 DESCRIPTION

34366 The *pthread_mutexattr_getprotocol()* and *pthread_mutexattr_setprotocol()* functions, respectively,
 34367 shall get and set the protocol attribute of a mutex attributes object pointed to by *attr* which was
 34368 previously created by the function *pthread_mutexattr_init()*.

34369 The *protocol* attribute defines the protocol to be followed in utilizing mutexes. The value of
 34370 *protocol* may be one of:

34371 PTHREAD_PRIO_NONE
 34372 TPI PTHREAD_PRIO_INHERIT
 34373 TPP PTHREAD_PRIO_PROTECT
 34374

34375 which are defined in the <pthread.h> header.

34376 When a thread owns a mutex with the PTHREAD_PRIO_NONE *protocol* attribute, its priority
 34377 and scheduling shall not be affected by its mutex ownership.

34378 TPI When a thread is blocking higher priority threads because of owning one or more mutexes with
 34379 the PTHREAD_PRIO_INHERIT *protocol* attribute, it shall execute at the higher of its priority or
 34380 the priority of the highest priority thread waiting on any of the mutexes owned by this thread
 34381 and initialized with this protocol.

34382 TPP When a thread owns one or more mutexes initialized with the PTHREAD_PRIO_PROTECT
 34383 protocol, it shall execute at the higher of its priority or the highest of the priority ceilings of all
 34384 the mutexes owned by this thread and initialized with this attribute, regardless of whether other
 34385 threads are blocked on any of these mutexes or not.

34386 While a thread is holding a mutex which has been initialized with the
 34387 PTHREAD_PRIO_INHERIT or PTHREAD_PRIO_PROTECT protocol attributes, it shall not be
 34388 subject to being moved to the tail of the scheduling queue at its priority in the event that its
 34389 original priority is changed, such as by a call to *sched_setparam()*. Likewise, when a thread
 34390 unlocks a mutex that has been initialized with the PTHREAD_PRIO_INHERIT or
 34391 PTHREAD_PRIO_PROTECT protocol attributes, it shall not be subject to being moved to the tail
 34392 of the scheduling queue at its priority in the event that its original priority is changed.

34393 If a thread simultaneously owns several mutexes initialized with different protocols, it shall
 34394 execute at the highest of the priorities that it would have obtained by each of these protocols.

34395 TPI When a thread makes a call to *pthread_mutex_lock()*, the mutex was initialized with the protocol
 34396 attribute having the value PTHREAD_PRIO_INHERIT, when the calling thread is blocked
 34397 because the mutex is owned by another thread, that owner thread shall inherit the priority level
 34398 of the calling thread as long as it continues to own the mutex. The implementation shall update
 34399 its execution priority to the maximum of its assigned priority and all its inherited priorities.
 34400 Furthermore, if this owner thread itself becomes blocked on another mutex, the same priority

34401 inheritance effect shall be propagated to this other owner thread, in a recursive manner.

34402 RETURN VALUE

34403 Upon successful completion, the *pthread_mutexattr_getprotocol()* and
 34404 *pthread_mutexattr_setprotocol()* functions shall return zero; otherwise, an error number shall be
 34405 returned to indicate the error.

34406 ERRORS

34407 The *pthread_mutexattr_setprotocol()* function shall fail if:

34408 [ENOTSUP] The value specified by *protocol* is an unsupported value.

34409 The *pthread_mutexattr_getprotocol()* and *pthread_mutexattr_setprotocol()* functions may fail if:

34410 [EINVAL] The value specified by *attr* or *protocol* is invalid.

34411 [EPERM] The caller does not have the privilege to perform the operation.

34412 These functions shall not return an error code of [EINTR].

34413 EXAMPLES

34414 None.

34415 APPLICATION USAGE

34416 None.

34417 RATIONALE

34418 None.

34419 FUTURE DIRECTIONS

34420 None.

34421 SEE ALSO

34422 *pthread_cond_destroy()*, *pthread_create()*, *pthread_mutex_destroy()*, the Base Definitions volume of
 34423 IEEE Std 1003.1-2001, <pthread.h>

34424 CHANGE HISTORY

34425 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34426 Marked as part of the Realtime Threads Feature Group.

34427 Issue 6

34428 The *pthread_mutexattr_getprotocol()* and *pthread_mutexattr_setprotocol()* functions are marked as
 34429 part of the Threads option and either the Thread Priority Protection or Thread Priority
 34430 Inheritance options.

34431 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 34432 implementation does not support the Thread Priority Protection or Thread Priority Inheritance
 34433 options.

34434 The **restrict** keyword is added to the *pthread_mutexattr_getprotocol()* prototype for alignment
 34435 with the ISO/IEC 9899:1999 standard.

34436 NAME

34437 pthread_mutexattr_getpshared, pthread_mutexattr_setpshared — get and set the process-
 34438 shared attribute

34439 SYNOPSIS

34440 THR TSH #include <pthread.h>

```
34441 int pthread_mutexattr_getpshared(const pthread_mutexattr_t *
34442     restrict attr, int *restrict pshared);
34443 int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
34444     int pshared);
34445
```

34446 DESCRIPTION

34447 The *pthread_mutexattr_getpshared()* function shall obtain the value of the *process-shared* attribute
 34448 from the attributes object referenced by *attr*. The *pthread_mutexattr_setpshared()* function shall
 34449 set the *process-shared* attribute in an initialized attributes object referenced by *attr*.

34450 The *process-shared* attribute is set to PTHREAD_PROCESS_SHARED to permit a mutex to be
 34451 operated upon by any thread that has access to the memory where the mutex is allocated, even if
 34452 the mutex is allocated in memory that is shared by multiple processes. If the *process-shared*
 34453 attribute is PTHREAD_PROCESS_PRIVATE, the mutex shall only be operated upon by threads
 34454 created within the same process as the thread that initialized the mutex; if threads of differing
 34455 processes attempt to operate on such a mutex, the behavior is undefined. The default value of
 34456 the attribute shall be PTHREAD_PROCESS_PRIVATE.

34457 RETURN VALUE

34458 Upon successful completion, *pthread_mutexattr_setpshared()* shall return zero; otherwise, an error
 34459 number shall be returned to indicate the error.

34460 Upon successful completion, *pthread_mutexattr_getpshared()* shall return zero and store the value
 34461 of the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter.
 34462 Otherwise, an error number shall be returned to indicate the error.

34463 ERRORS

34464 The *pthread_mutexattr_getpshared()* and *pthread_mutexattr_setpshared()* functions may fail if:

34465 [EINVAL] The value specified by *attr* is invalid.

34466 The *pthread_mutexattr_setpshared()* function may fail if:

34467 [EINVAL] The new value specified for the attribute is outside the range of legal values
 34468 for that attribute.

34469 These functions shall not return an error code of [EINTR].

34470 EXAMPLES

34471 None.

34472 APPLICATION USAGE

34473 None.

34474 RATIONALE

34475 None.

34476 FUTURE DIRECTIONS

34477 None.

34478 **SEE ALSO**

34479 *pthread_cond_destroy()*, *pthread_create()*, *pthread_mutex_destroy()*, *pthread_mutexattr_destroy()*, the
34480 Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

34481 **CHANGE HISTORY**

34482 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34483 **Issue 6**

34484 The *pthread_mutexattr_getpshared()* and *pthread_mutexattr_setpshared()* functions are marked as
34485 part of the Threads and Thread Process-Shared Synchronization options.

34486 The **restrict** keyword is added to the *pthread_mutexattr_getpshared()* prototype for alignment
34487 with the ISO/IEC 9899:1999 standard.

34488 **NAME**

34489 pthread_mutexattr_gettype, pthread_mutexattr_settype — get and set the mutex type attribute

34490 **SYNOPSIS**34491 XSI

```
#include <pthread.h>
```

34492

```
int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,  
34493 int *restrict type);
```

34494

```
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

34495

34496 **DESCRIPTION**34497 The *pthread_mutexattr_gettype()* and *pthread_mutexattr_settype()* functions, respectively, shall get
34498 and set the mutex *type* attribute. This attribute is set in the *type* parameter to these functions. The
34499 default value of the *type* attribute is PTHREAD_MUTEX_DEFAULT.34500 The type of mutex is contained in the *type* attribute of the mutex attributes. Valid mutex types
34501 include:

34502 PTHREAD_MUTEX_NORMAL

34503 This type of mutex does not detect deadlock. A thread attempting to relock this mutex
34504 without first unlocking it shall deadlock. Attempting to unlock a mutex locked by a
34505 different thread results in undefined behavior. Attempting to unlock an unlocked mutex
34506 results in undefined behavior.

34507 PTHREAD_MUTEX_ERRORCHECK

34508 This type of mutex provides error checking. A thread attempting to relock this mutex
34509 without first unlocking it shall return with an error. A thread attempting to unlock a mutex
34510 which another thread has locked shall return with an error. A thread attempting to unlock
34511 an unlocked mutex shall return with an error.

34512 PTHREAD_MUTEX_RECURSIVE

34513 A thread attempting to relock this mutex without first unlocking it shall succeed in locking
34514 the mutex. The relocking deadlock which can occur with mutexes of type
34515 PTHREAD_MUTEX_NORMAL cannot occur with this type of mutex. Multiple locks of this
34516 mutex shall require the same number of unlocks to release the mutex before another thread
34517 can acquire the mutex. A thread attempting to unlock a mutex which another thread has
34518 locked shall return with an error. A thread attempting to unlock an unlocked mutex shall
34519 return with an error.

34520 PTHREAD_MUTEX_DEFAULT

34521 Attempting to recursively lock a mutex of this type results in undefined behavior.
34522 Attempting to unlock a mutex of this type which was not locked by the calling thread
34523 results in undefined behavior. Attempting to unlock a mutex of this type which is not
34524 locked results in undefined behavior. An implementation may map this mutex to one of the
34525 other mutex types.34526 **RETURN VALUE**34527 Upon successful completion, the *pthread_mutexattr_gettype()* function shall return zero and store
34528 the value of the *type* attribute of *attr* into the object referenced by the *type* parameter. Otherwise,
34529 an error shall be returned to indicate the error.34530 If successful, the *pthread_mutexattr_settype()* function shall return zero; otherwise, an error
34531 number shall be returned to indicate the error.

34532 **ERRORS**

34533 The *pthread_mutexattr_settype()* function shall fail if:

34534 [EINVAL] The value *type* is invalid.

34535 The *pthread_mutexattr_gettype()* and *pthread_mutexattr_settype()* functions may fail if:

34536 [EINVAL] The value specified by *attr* is invalid.

34537 These functions shall not return an error code of [EINTR].

34538 **EXAMPLES**

34539 None.

34540 **APPLICATION USAGE**

34541 It is advised that an application should not use a PTHREAD_MUTEX_RECURSIVE mutex with
34542 condition variables because the implicit unlock performed for a *pthread_cond_timedwait()* or
34543 *pthread_cond_wait()* may not actually release the mutex (if it had been locked multiple times). If
34544 this happens, no other thread can satisfy the condition of the predicate.

34545 **RATIONALE**

34546 None.

34547 **FUTURE DIRECTIONS**

34548 None.

34549 **SEE ALSO**

34550 *pthread_cond_timedwait()*, the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

34551 **CHANGE HISTORY**

34552 First released in Issue 5.

34553 **Issue 6**

34554 The Open Group Corrigendum U033/3 is applied. The SYNOPSIS for
34555 *pthread_mutexattr_gettype()* is updated so that the first argument is of type **const**
34556 **pthread_mutexattr_t** *.

34557 The **restrict** keyword is added to the *pthread_mutexattr_gettype()* prototype for alignment with
34558 the ISO/IEC 9899:1999 standard.

34559 NAME

34560 pthread_mutexattr_init — initialize the mutex attributes object

34561 SYNOPSIS

34562 THR `#include <pthread.h>`

34563 `int pthread_mutexattr_init(pthread_mutexattr_t *attr);`

34564

34565 DESCRIPTION

34566 Refer to *pthread_mutexattr_destroy()*.

34567 **NAME**

34568 pthread_mutexattr_setprioceiling — set the prioceiling attribute of the mutex attributes object
34569 (**REALTIME THREADS**)

34570 **SYNOPSIS**

34571 THR TPP #include <pthread.h>

34572 int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
34573 int prioceiling);

34574

34575 **DESCRIPTION**

34576 Refer to *pthread_mutexattr_getprioceiling()*.

34577 NAME

34578 pthread_mutexattr_setprotocol — set the protocol attribute of the mutex attributes object
34579 (REALTIME THREADS)

34580 SYNOPSIS

34581 THR `#include <pthread.h>`

34582 TPP|TPI `int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,`
34583 `int protocol);`

34584

34585 DESCRIPTION

34586 Refer to *pthread_mutexattr_getprotocol()*.

34587 **NAME**

34588 pthread_mutexattr_setpshared — set the process-shared attribute

34589 **SYNOPSIS**

34590 THR TSH #include <pthread.h>

34591 int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
34592 int pshared);

34593

34594 **DESCRIPTION**34595 Refer to *pthread_mutexattr_getpshared()*.

34596 NAME

34597 pthread_mutexattr_settype — set the mutex type attribute

34598 SYNOPSIS

34599 XSI `#include <pthread.h>`

34600 `int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);`

34601

34602 DESCRIPTION

34603 Refer to *pthread_mutexattr_gettype()*.

34604 **NAME**

34605 pthread_once — dynamic package initialization

34606 **SYNOPSIS**

```

34607 THR    #include <pthread.h>

34608        int pthread_once(pthread_once_t *once_control,
34609                          void (*init_routine)(void));
34610        pthread_once_t once_control = PTHREAD_ONCE_INIT;
34611

```

34612 **DESCRIPTION**

34613 The first call to *pthread_once()* by any thread in a process, with a given *once_control*, shall call the
 34614 *init_routine* with no arguments. Subsequent calls of *pthread_once()* with the same *once_control*
 34615 shall not call the *init_routine*. On return from *pthread_once()*, *init_routine* shall have completed.
 34616 The *once_control* parameter shall determine whether the associated initialization routine has
 34617 been called.

34618 The *pthread_once()* function is not a cancellation point. However, if *init_routine* is a cancellation
 34619 point and is canceled, the effect on *once_control* shall be as if *pthread_once()* was never called.

34620 The constant PTHREAD_ONCE_INIT is defined in the **<pthread.h>** header.

34621 The behavior of *pthread_once()* is undefined if *once_control* has automatic storage duration or is
 34622 not initialized by PTHREAD_ONCE_INIT.

34623 **RETURN VALUE**

34624 Upon successful completion, *pthread_once()* shall return zero; otherwise, an error number shall
 34625 be returned to indicate the error.

34626 **ERRORS**

34627 The *pthread_once()* function may fail if:

34628 [EINVAL] If either *once_control* or *init_routine* is invalid.

34629 The *pthread_once()* function shall not return an error code of [EINTR].

34630 **EXAMPLES**

34631 None.

34632 **APPLICATION USAGE**

34633 None.

34634 **RATIONALE**

34635 Some C libraries are designed for dynamic initialization. That is, the global initialization for the
 34636 library is performed when the first procedure in the library is called. In a single-threaded
 34637 program, this is normally implemented using a static variable whose value is checked on entry
 34638 to a routine, as follows:

```

34639 static int random_is_initialized = 0;
34640 extern int initialize_random();

34641 int random_function()
34642 {
34643     if (random_is_initialized == 0) {
34644         initialize_random();
34645         random_is_initialized = 1;
34646     }
34647     ... /* Operations performed after initialization. */
34648 }

```


34649 To keep the same structure in a multi-threaded program, a new primitive is needed. Otherwise,
34650 library initialization has to be accomplished by an explicit call to a library-exported initialization
34651 function prior to any use of the library.

34652 For dynamic library initialization in a multi-threaded process, a simple initialization flag is not
34653 sufficient; the flag needs to be protected against modification by multiple threads
34654 simultaneously calling into the library. Protecting the flag requires the use of a mutex; however,
34655 mutexes have to be initialized before they are used. Ensuring that the mutex is only initialized
34656 once requires a recursive solution to this problem.

34657 The use of *pthread_once()* not only supplies an implementation-guaranteed means of dynamic
34658 initialization, it provides an aid to the reliable construction of multi-threaded and realtime
34659 systems. The preceding example then becomes:

```
34660 #include <pthread.h>
34661 static pthread_once_t random_is_initialized = PTHREAD_ONCE_INIT;
34662 extern int initialize_random();

34663 int random_function()
34664 {
34665     (void) pthread_once(&random_is_initialized, initialize_random);
34666     ... /* Operations performed after initialization. */
34667 }
```

34668 Note that a **pthread_once_t** cannot be an array because some compilers do not accept the
34669 construct **&<array_name>**.

34670 FUTURE DIRECTIONS

34671 None.

34672 SEE ALSO

34673 The Base Definitions volume of IEEE Std 1003.1-2001, **<pthread.h>**

34674 CHANGE HISTORY

34675 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34676 Issue 6

34677 The *pthread_once()* function is marked as part of the Threads option.

34678 The [EINVAL] error is added as a may fail case for if either argument is invalid.

34679 **NAME**

34680 pthread_rwlock_destroy, pthread_rwlock_init — destroy and initialize a read-write lock object

34681 **SYNOPSIS**

34682 THR #include <pthread.h>

```

34683 int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
34684 int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
34685     const pthread_rwlockattr_t *restrict attr);
34686 
```

34687 **DESCRIPTION**

34688 The *pthread_rwlock_destroy()* function shall destroy the read-write lock object referenced by
 34689 *rwlock* and release any resources used by the lock. The effect of subsequent use of the lock is
 34690 undefined until the lock is reinitialized by another call to *pthread_rwlock_init()*. An
 34691 implementation may cause *pthread_rwlock_destroy()* to set the object referenced by *rwlock* to an
 34692 invalid value. Results are undefined if *pthread_rwlock_destroy()* is called when any thread holds
 34693 *rwlock*. Attempting to destroy an uninitialized read-write lock results in undefined behavior.

34694 The *pthread_rwlock_init()* function shall allocate any resources required to use the read-write
 34695 lock referenced by *rwlock* and initializes the lock to an unlocked state with attributes referenced
 34696 by *attr*. If *attr* is NULL, the default read-write lock attributes shall be used; the effect is the same
 34697 as passing the address of a default read-write lock attributes object. Once initialized, the lock can
 34698 be used any number of times without being reinitialized. Results are undefined if
 34699 *pthread_rwlock_init()* is called specifying an already initialized read-write lock. Results are
 34700 undefined if a read-write lock is used without first being initialized.

34701 If the *pthread_rwlock_init()* function fails, *rwlock* shall not be initialized and the contents of *rwlock*
 34702 are undefined.

34703 Only the object referenced by *rwlock* may be used for performing synchronization. The result of
 34704 referring to copies of that object in calls to *pthread_rwlock_destroy()*, *pthread_rwlock_rdlock()*,
 34705 *pthread_rwlock_timedrdlock()*, *pthread_rwlock_timedwrlock()*, *pthread_rwlock_tryrdlock()*,
 34706 *pthread_rwlock_trywrlock()*, *pthread_rwlock_unlock()*, or *pthread_rwlock_wrlock()* is undefined.

34707 **RETURN VALUE**

34708 If successful, the *pthread_rwlock_destroy()* and *pthread_rwlock_init()* functions shall return zero;
 34709 otherwise, an error number shall be returned to indicate the error.

34710 The [EBUSY] and [EINVAL] error checks, if implemented, act as if they were performed
 34711 immediately at the beginning of processing for the function and caused an error return prior to
 34712 modifying the state of the read-write lock specified by *rwlock*.

34713 **ERRORS**

34714 The *pthread_rwlock_destroy()* function may fail if:

34715 [EBUSY] The implementation has detected an attempt to destroy the object referenced
 34716 by *rwlock* while it is locked.

34717 [EINVAL] The value specified by *rwlock* is invalid.

34718 The *pthread_rwlock_init()* function shall fail if:

34719 [EAGAIN] The system lacked the necessary resources (other than memory) to initialize
 34720 another read-write lock.

34721 [ENOMEM] Insufficient memory exists to initialize the read-write lock.

34722 [EPERM] The caller does not have the privilege to perform the operation.

34723 The *pthread_rwlock_init()* function may fail if:

34724 [EBUSY] The implementation has detected an attempt to reinitialize the object
34725 referenced by *rwlock*, a previously initialized but not yet destroyed read-write
34726 lock.

34727 [EINVAL] The value specified by *attr* is invalid.

34728 These functions shall not return an error code of [EINTR].

34729 EXAMPLES

34730 None.

34731 APPLICATION USAGE

34732 None.

34733 RATIONALE

34734 None.

34735 FUTURE DIRECTIONS

34736 None.

34737 SEE ALSO

34738 *pthread_rwlock_rdlock()*, *pthread_rwlock_timedrdlock()*, *pthread_rwlock_timedwrlock()*,
34739 *pthread_rwlock_tryrdlock()*, *pthread_rwlock_trywrlock()*, *pthread_rwlock_unlock()*,
34740 *pthread_rwlock_wrlock()*, the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

34741 CHANGE HISTORY

34742 First released in Issue 5.

34743 Issue 6

34744 The following changes are made for alignment with IEEE Std 1003.1j-2000:

- 34745 • The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is
34746 now part of the Threads option (previously it was part of the Read-Write Locks option in
34747 IEEE Std 1003.1j-2000 and also part of the XSI extension). The initializer macro is also deleted
34748 from the SYNOPSIS.
- 34749 • The DESCRIPTION is updated as follows:
 - 34750 — It explicitly notes allocation of resources upon initialization of a read-write lock object.
 - 34751 — A paragraph is added specifying that copies of read-write lock objects may not be used.
- 34752 • An [EINVAL] error is added to the ERRORS section for *pthread_rwlock_init()*, indicating that
34753 the *rwlock* value is invalid.
- 34754 • The SEE ALSO section is updated.

34755 The **restrict** keyword is added to the *pthread_rwlock_init()* prototype for alignment with the
34756 ISO/IEC 9899:1999 standard.

34757 **NAME**

34758 pthread_rwlock_rdlock, pthread_rwlock_tryrdlock — lock a read-write lock object for reading

34759 **SYNOPSIS**

34760 THR #include <pthread.h>

34761 int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);

34762 int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);

34763

34764 **DESCRIPTION**

34765 The *pthread_rwlock_rdlock()* function shall apply a read lock to the read-write lock referenced by
 34766 *rwlock*. The calling thread acquires the read lock if a writer does not hold the lock and there are
 34767 no writers blocked on the lock.

34768 TPS If the Thread Execution Scheduling option is supported, and the threads involved in the lock are
 34769 executing with the scheduling policies SCHED_FIFO or SCHED_RR, the calling thread shall not
 34770 acquire the lock if a writer holds the lock or if writers of higher or equal priority are blocked on
 34771 the lock; otherwise, the calling thread shall acquire the lock.

34772 TPS TSP If the Threads Execution Scheduling option is supported, and the threads involved in the lock
 34773 are executing with the SCHED_SPORADIC scheduling policy, the calling thread shall not
 34774 acquire the lock if a writer holds the lock or if writers of higher or equal priority are blocked on
 34775 the lock; otherwise, the calling thread shall acquire the lock.

34776 If the Thread Execution Scheduling option is not supported, it is implementation-defined
 34777 whether the calling thread acquires the lock when a writer does not hold the lock and there are
 34778 writers blocked on the lock. If a writer holds the lock, the calling thread shall not acquire the
 34779 read lock. If the read lock is not acquired, the calling thread shall block until it can acquire the
 34780 lock. The calling thread may deadlock if at the time the call is made it holds a write lock.

34781 A thread may hold multiple concurrent read locks on *rwlock* (that is, successfully call the
 34782 *pthread_rwlock_rdlock()* function *n* times). If so, the application shall ensure that the thread
 34783 performs matching unlocks (that is, it calls the *pthread_rwlock_unlock()* function *n* times).

34784 The maximum number of simultaneous read locks that an implementation guarantees can be
 34785 applied to a read-write lock shall be implementation-defined. The *pthread_rwlock_rdlock()*
 34786 function may fail if this maximum would be exceeded.

34787 The *pthread_rwlock_tryrdlock()* function shall apply a read lock as in the *pthread_rwlock_rdlock()*
 34788 function, with the exception that the function shall fail if the equivalent *pthread_rwlock_rdlock()*
 34789 call would have blocked the calling thread. In no case shall the *pthread_rwlock_tryrdlock()*
 34790 function ever block; it always either acquires the lock or fails and returns immediately.

34791 Results are undefined if any of these functions are called with an uninitialized read-write lock.

34792 If a signal is delivered to a thread waiting for a read-write lock for reading, upon return from the
 34793 signal handler the thread resumes waiting for the read-write lock for reading as if it was not
 34794 interrupted.

34795 **RETURN VALUE**

34796 If successful, the *pthread_rwlock_rdlock()* function shall return zero; otherwise, an error number
 34797 shall be returned to indicate the error.

34798 The *pthread_rwlock_tryrdlock()* function shall return zero if the lock for reading on the read-write
 34799 lock object referenced by *rwlock* is acquired. Otherwise, an error number shall be returned to
 34800 indicate the error.

34801 **ERRORS**

34802 The *pthread_rwlock_tryrdlock()* function shall fail if:

34803 [EBUSY] The read-write lock could not be acquired for reading because a writer holds
34804 the lock or a writer with the appropriate priority was blocked on it.

34805 The *pthread_rwlock_rdlock()* and *pthread_rwlock_tryrdlock()* functions may fail if:

34806 [EINVAL] The value specified by *rwlock* does not refer to an initialized read-write lock
34807 object.

34808 [EAGAIN] The read lock could not be acquired because the maximum number of read
34809 locks for *rwlock* has been exceeded.

34810 The *pthread_rwlock_rdlock()* function may fail if:

34811 [EDEADLK] The current thread already owns the read-write lock for writing.

34812 These functions shall not return an error code of [EINTR].

34813 **EXAMPLES**

34814 None.

34815 **APPLICATION USAGE**

34816 Applications using these functions may be subject to priority inversion, as discussed in the Base
34817 Definitions volume of IEEE Std 1003.1-2001, Section 3.285, Priority Inversion.

34818 **RATIONALE**

34819 None.

34820 **FUTURE DIRECTIONS**

34821 None.

34822 **SEE ALSO**

34823 *pthread_rwlock_destroy()*, *pthread_rwlock_timedrdlock()*, *pthread_rwlock_timedwrlock()*,
34824 *pthread_rwlock_trywrlock()*, *pthread_rwlock_unlock()*, *pthread_rwlock_wrlock()*, the Base Definitions
34825 volume of IEEE Std 1003.1-2001, <pthread.h>

34826 **CHANGE HISTORY**

34827 First released in Issue 5.

34828 **Issue 6**

34829 The following changes are made for alignment with IEEE Std 1003.1j-2000:

- 34830 • The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is
34831 now part of the Threads option (previously it was part of the Read-Write Locks option in
34832 IEEE Std 1003.1j-2000 and also part of the XSI extension).
- 34833 • The DESCRIPTION is updated as follows:
 - 34834 — Conditions under which writers have precedence over readers are specified.
 - 34835 — Failure of *pthread_rwlock_tryrdlock()* is clarified.
 - 34836 — A paragraph on the maximum number of read locks is added.
- 34837 • In the ERRORS sections, [EBUSY] is modified to take into account write priority, and
34838 [EDEADLK] is deleted as a *pthread_rwlock_tryrdlock()* error.
- 34839 • The SEE ALSO section is updated.

34840 **NAME**

34841 pthread_rwlock_timedrdlock — lock a read-write lock for reading

34842 **SYNOPSIS**

34843 THR TMO #include <pthread.h>

34844 #include <time.h>

34845 int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict *rwlock*,34846 const struct timespec *restrict *abs_timeout*);

34847

34848 **DESCRIPTION**

34849 The *pthread_rwlock_timedrdlock()* function shall apply a read lock to the read-write lock
 34850 referenced by *rwlock* as in the *pthread_rwlock_rdlock()* function. However, if the lock cannot be
 34851 acquired without waiting for other threads to unlock the lock, this wait shall be terminated
 34852 when the specified timeout expires. The timeout shall expire when the absolute time specified
 34853 by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the
 34854 value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout*
 34855 has already been passed at the time of the call.

34856 TMR If the Timers option is supported, the timeout shall be based on the `CLOCK_REALTIME` clock. If
 34857 the Timers option is not supported, the timeout shall be based on the system clock as returned
 34858 by the *time()* function. The resolution of the timeout shall be the resolution of the clock on which
 34859 it is based. The **timespec** data type is defined in the `<time.h>` header. Under no circumstances
 34860 shall the function fail with a timeout if the lock can be acquired immediately. The validity of the
 34861 *abs_timeout* parameter need not be checked if the lock can be immediately acquired.

34862 If a signal that causes a signal handler to be executed is delivered to a thread blocked on a read-
 34863 write lock via a call to *pthread_rwlock_timedrdlock()*, upon return from the signal handler the
 34864 thread shall resume waiting for the lock as if it was not interrupted.

34865 The calling thread may deadlock if at the time the call is made it holds a write lock on *rwlock*.
 34866 The results are undefined if this function is called with an uninitialized read-write lock.

34867 **RETURN VALUE**

34868 The *pthread_rwlock_timedrdlock()* function shall return zero if the lock for reading on the read-
 34869 write lock object referenced by *rwlock* is acquired. Otherwise, an error number shall be returned
 34870 to indicate the error.

34871 **ERRORS**34872 The *pthread_rwlock_timedrdlock()* function shall fail if:

34873 [ETIMEDOUT] The lock could not be acquired before the specified timeout expired.

34874 The *pthread_rwlock_timedrdlock()* function may fail if:

34875 [EAGAIN] The read lock could not be acquired because the maximum number of read
 34876 locks for lock would be exceeded.

34877 [EDEADLK] The calling thread already holds a write lock on *rwlock*.

34878 [EINVAL] The value specified by *rwlock* does not refer to an initialized read-write lock
 34879 object, or the *abs_timeout* nanosecond value is less than zero or greater than or
 34880 equal to 1 000 million.

34881 This function shall not return an error code of [EINTR].

34882 EXAMPLES

34883 None.

34884 APPLICATION USAGE

34885 Applications using this function may be subject to priority inversion, as discussed in the Base
34886 Definitions volume of IEEE Std 1003.1-2001, Section 3.285, Priority Inversion.

34887 The *pthread_rwlock_timedrdlock()* function is part of the Threads and Timeouts options and need
34888 not be provided on all implementations.

34889 RATIONALE

34890 None.

34891 FUTURE DIRECTIONS

34892 None.

34893 SEE ALSO

34894 *pthread_rwlock_destroy()*, *pthread_rwlock_rdlock()*, *pthread_rwlock_timedwrlock()*,
34895 *pthread_rwlock_tryrdlock()*, *pthread_rwlock_trywrlock()*, *pthread_rwlock_unlock()*,
34896 *pthread_rwlock_wrlock()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<pthread.h>**,
34897 **<time.h>**

34898 CHANGE HISTORY

34899 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

34900 **NAME**

34901 pthread_rwlock_timedwrlock — lock a read-write lock for writing

34902 **SYNOPSIS**

34903 THR TMO #include <pthread.h>

34904 #include <time.h>

34905 int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict *rwlock*,34906 const struct timespec *restrict *abs_timeout*);

34907

34908 **DESCRIPTION**

34909 The *pthread_rwlock_timedwrlock()* function shall apply a write lock to the read-write lock
 34910 referenced by *rwlock* as in the *pthread_rwlock_wrlock()* function. However, if the lock cannot be
 34911 acquired without waiting for other threads to unlock the lock, this wait shall be terminated
 34912 when the specified timeout expires. The timeout shall expire when the absolute time specified
 34913 by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the
 34914 value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout*
 34915 has already been passed at the time of the call.

34916 TMR If the Timers option is supported, the timeout shall be based on the `CLOCK_REALTIME` clock. If
 34917 the Timers option is not supported, the timeout shall be based on the system clock as returned
 34918 by the *time()* function. The resolution of the timeout shall be the resolution of the clock on which
 34919 it is based. The **timespec** data type is defined in the `<time.h>` header. Under no circumstances
 34920 shall the function fail with a timeout if the lock can be acquired immediately. The validity of the
 34921 *abs_timeout* parameter need not be checked if the lock can be immediately acquired.

34922 If a signal that causes a signal handler to be executed is delivered to a thread blocked on a read-
 34923 write lock via a call to *pthread_rwlock_timedwrlock()*, upon return from the signal handler the
 34924 thread shall resume waiting for the lock as if it was not interrupted.

34925 The calling thread may deadlock if at the time the call is made it holds the read-write lock. The
 34926 results are undefined if this function is called with an uninitialized read-write lock.

34927 **RETURN VALUE**

34928 The *pthread_rwlock_timedwrlock()* function shall return zero if the lock for writing on the read-
 34929 write lock object referenced by *rwlock* is acquired. Otherwise, an error number shall be returned
 34930 to indicate the error.

34931 **ERRORS**34932 The *pthread_rwlock_timedwrlock()* function shall fail if:

34933 [ETIMEDOUT] The lock could not be acquired before the specified timeout expired.

34934 The *pthread_rwlock_timedwrlock()* function may fail if:34935 [EDEADLK] The calling thread already holds the *rwlock*.

34936 [EINVAL] The value specified by *rwlock* does not refer to an initialized read-write lock
 34937 object, or the *abs_timeout* nanosecond value is less than zero or greater than or
 34938 equal to 1 000 million.

34939 This function shall not return an error code of [EINTR].

34940 EXAMPLES

34941 None.

34942 APPLICATION USAGE

34943 Applications using this function may be subject to priority inversion, as discussed in the Base
34944 Definitions volume of IEEE Std 1003.1-2001, Section 3.285, Priority Inversion.

34945 The *pthread_rwlock_timedwrlock()* function is part of the Threads and Timeouts options and need
34946 not be provided on all implementations.

34947 RATIONALE

34948 None.

34949 FUTURE DIRECTIONS

34950 None.

34951 SEE ALSO

34952 *pthread_rwlock_destroy()*, *pthread_rwlock_rdlock()*, *pthread_rwlock_timedrdlock()*,
34953 *pthread_rwlock_tryrdlock()*, *pthread_rwlock_trywrlock()*, *pthread_rwlock_unlock()*,
34954 *pthread_rwlock_wrlock()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>,
34955 <**time.h**>

34956 CHANGE HISTORY

34957 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

34958 **NAME**

34959 pthread_rwlock_tryrdlock — lock a read-write lock object for reading

34960 **SYNOPSIS**34961 THR `#include <pthread.h>`34962 `int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);`

34963

34964 **DESCRIPTION**34965 Refer to *pthread_rwlock_rdlock()*.

34966 **NAME**

34967 pthread_rwlock_trywrlock, pthread_rwlock_wrlock — lock a read-write lock object for writing

34968 **SYNOPSIS**34969 THR `#include <pthread.h>`34970 `int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);`34971 `int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);`

34972

34973 **DESCRIPTION**

34974 The *pthread_rwlock_trywrlock()* function shall apply a write lock like the *pthread_rwlock_wrlock()*
 34975 function, with the exception that the function shall fail if any thread currently holds *rwlock* (for
 34976 reading or writing).

34977 The *pthread_rwlock_wrlock()* function shall apply a write lock to the read-write lock referenced
 34978 by *rwlock*. The calling thread acquires the write lock if no other thread (reader or writer) holds
 34979 the read-write lock *rwlock*. Otherwise, the thread shall block until it can acquire the lock. The
 34980 calling thread may deadlock if at the time the call is made it holds the read-write lock (whether a
 34981 read or write lock).

34982 Implementations may favor writers over readers to avoid writer starvation.

34983 Results are undefined if any of these functions are called with an uninitialized read-write lock.

34984 If a signal is delivered to a thread waiting for a read-write lock for writing, upon return from the
 34985 signal handler the thread resumes waiting for the read-write lock for writing as if it was not
 34986 interrupted.

34987 **RETURN VALUE**

34988 The *pthread_rwlock_trywrlock()* function shall return zero if the lock for writing on the read-write
 34989 lock object referenced by *rwlock* is acquired. Otherwise, an error number shall be returned to
 34990 indicate the error.

34991 If successful, the *pthread_rwlock_wrlock()* function shall return zero; otherwise, an error number
 34992 shall be returned to indicate the error.

34993 **ERRORS**

34994 The *pthread_rwlock_trywrlock()* function shall fail if:

34995 [EBUSY] The read-write lock could not be acquired for writing because it was already
 34996 locked for reading or writing.

34997 The *pthread_rwlock_trywrlock()* and *pthread_rwlock_wrlock()* functions may fail if:

34998 [EINVAL] The value specified by *rwlock* does not refer to an initialized read-write lock
 34999 object.

35000 The *pthread_rwlock_wrlock()* function may fail if:

35001 [EDEADLK] The current thread already owns the read-write lock for writing or reading.

35002 These functions shall not return an error code of [EINTR].

35003 **EXAMPLES**

35004 None.

35005 **APPLICATION USAGE**

35006 Applications using these functions may be subject to priority inversion, as discussed in the Base
35007 Definitions volume of IEEE Std 1003.1-2001, Section 3.285, Priority Inversion.

35008 **RATIONALE**

35009 None.

35010 **FUTURE DIRECTIONS**

35011 None.

35012 **SEE ALSO**

35013 *pthread_rwlock_destroy()*, *pthread_rwlock_rdlock()*, *pthread_rwlock_timedrdlock()*,
35014 *pthread_rwlock_timedwrlock()*, *pthread_rwlock_tryrdlock()*, *pthread_rwlock_unlock()*, the Base
35015 Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

35016 **CHANGE HISTORY**

35017 First released in Issue 5.

35018 **Issue 6**

35019 The following changes are made for alignment with IEEE Std 1003.1j-2000:

- 35020 • The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is
- 35021 now part of the Threads option (previously it was part of the Read-Write Locks option in
- 35022 IEEE Std 1003.1j-2000 and also part of the XSI extension).
- 35023 • The [EDEADLK] error is deleted as a *pthread_rwlock_trywrlock()* error.
- 35024 • The SEE ALSO section is updated.

35025 **NAME**

35026 pthread_rwlock_unlock — unlock a read-write lock object

35027 **SYNOPSIS**

35028 THR #include <pthread.h>

35029 int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);

35030

35031 **DESCRIPTION**

35032 The *pthread_rwlock_unlock()* function shall release a lock held on the read-write lock object
 35033 referenced by *rwlock*. Results are undefined if the read-write lock *rwlock* is not held by the
 35034 calling thread.

35035 If this function is called to release a read lock from the read-write lock object and there are other
 35036 read locks currently held on this read-write lock object, the read-write lock object remains in the
 35037 read locked state. If this function releases the last read lock for this read-write lock object, the
 35038 read-write lock object shall be put in the unlocked state with no owners.

35039 If this function is called to release a write lock for this read-write lock object, the read-write lock
 35040 object shall be put in the unlocked state.

35041 If there are threads blocked on the lock when it becomes available, the scheduling policy shall
 35042 TPS determine which thread(s) shall acquire the lock. If the Thread Execution Scheduling option is
 35043 supported, when threads executing with the scheduling policies SCHED_FIFO, SCHED_RR, or
 35044 SCHED_SPORADIC are waiting on the lock, they shall acquire the lock in priority order when
 35045 the lock becomes available. For equal priority threads, write locks shall take precedence over
 35046 read locks. If the Thread Execution Scheduling option is not supported, it is implementation-
 35047 defined whether write locks take precedence over read locks.

35048 Results are undefined if any of these functions are called with an uninitialized read-write lock.

35049 **RETURN VALUE**

35050 If successful, the *pthread_rwlock_unlock()* function shall return zero; otherwise, an error number
 35051 shall be returned to indicate the error.

35052 **ERRORS**

35053 The *pthread_rwlock_unlock()* function may fail if:

35054 [EINVAL] The value specified by *rwlock* does not refer to an initialized read-write lock
 35055 object.

35056 [EPERM] The current thread does not hold a lock on the read-write lock.

35057 The *pthread_rwlock_unlock()* function shall not return an error code of [EINTR].

35058 **EXAMPLES**

35059 None.

35060 **APPLICATION USAGE**

35061 None.

35062 **RATIONALE**

35063 None.

35064 **FUTURE DIRECTIONS**

35065 None.

35066 **SEE ALSO**

35067 *pthread_rwlock_destroy()*, *pthread_rwlock_rdlock()*, *pthread_rwlock_timedrdlock()*,
35068 *pthread_rwlock_timedwrlock()*, *pthread_rwlock_tryrdlock()*, *pthread_rwlock_trywrlock()*,
35069 *pthread_rwlock_wrlock()*, the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

35070 **CHANGE HISTORY**

35071 First released in Issue 5.

35072 **Issue 6**

35073 The following changes are made for alignment with IEEE Std 1003.1j-2000:

- 35074 • The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is
35075 now part of the Threads option (previously it was part of the Read-Write Locks option in
35076 IEEE Std 1003.1j-2000 and also part of the XSI extension).
- 35077 • The DESCRIPTION is updated as follows:
 - 35078 — The conditions under which writers have precedence over readers are specified.
 - 35079 — The concept of read-write lock owner is deleted.
- 35080 • The SEE ALSO section is updated.

35081 NAME

35082 pthread_rwlock_wrlock — lock a read-write lock object for writing

35083 SYNOPSIS

35084 THR `#include <pthread.h>`

35085 `int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);`

35086

35087 DESCRIPTION

35088 Refer to *pthread_rwlock_trywrlock()*.

35089 **NAME**

35090 pthread_rwlockattr_destroy, pthread_rwlockattr_init — destroy and initialize the read-write
 35091 lock attributes object

35092 **SYNOPSIS**

35093 THR `#include <pthread.h>`

35094 `int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);`

35095 `int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);`

35096

35097 **DESCRIPTION**

35098 The *pthread_rwlockattr_destroy()* function shall destroy a read-write lock attributes object. A
 35099 destroyed *attr* attributes object can be reinitialized using *pthread_rwlockattr_init()*; the results of
 35100 otherwise referencing the object after it has been destroyed are undefined. An implementation
 35101 may cause *pthread_rwlockattr_destroy()* to set the object referenced by *attr* to an invalid value.

35102 The *pthread_rwlockattr_init()* function shall initialize a read-write lock attributes object *attr* with
 35103 the default value for all of the attributes defined by the implementation.

35104 Results are undefined if *pthread_rwlockattr_init()* is called specifying an already initialized *attr*
 35105 attributes object.

35106 After a read-write lock attributes object has been used to initialize one or more read-write locks,
 35107 any function affecting the attributes object (including destruction) shall not affect any previously
 35108 initialized read-write locks.

35109 **RETURN VALUE**

35110 If successful, the *pthread_rwlockattr_destroy()* and *pthread_rwlockattr_init()* functions shall return
 35111 zero; otherwise, an error number shall be returned to indicate the error.

35112 **ERRORS**

35113 The *pthread_rwlockattr_destroy()* function may fail if:

35114 [EINVAL] The value specified by *attr* is invalid.

35115 The *pthread_rwlockattr_init()* function shall fail if:

35116 [ENOMEM] Insufficient memory exists to initialize the read-write lock attributes object.

35117 These functions shall not return an error code of [EINTR].

35118 **EXAMPLES**

35119 None.

35120 **APPLICATION USAGE**

35121 None.

35122 **RATIONALE**

35123 None.

35124 **FUTURE DIRECTIONS**

35125 None.

35126 **SEE ALSO**

35127 *pthread_rwlock_destroy()*, *pthread_rwlockattr_getpshared()*, *pthread_rwlockattr_setpshared()*, the
 35128 Base Definitions volume of IEEE Std 1003.1-2001, **<pthread.h>**

35129 **CHANGE HISTORY**

35130 First released in Issue 5.

35131 **Issue 6**

35132 The following changes are made for alignment with IEEE Std 1003.1j-2000:

- 35133 • The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is
35134 now part of the Threads option (previously it was part of the Read-Write Locks option in
35135 IEEE Std 1003.1j-2000 and also part of the XSI extension).
- 35136 • The SEE ALSO section is updated.

35137 **NAME**

35138 pthread_rwlockattr_getpshared, pthread_rwlockattr_setpshared — get and set the process-
 35139 shared attribute of the read-write lock attributes object

35140 **SYNOPSIS**

35141 THR TSH #include <pthread.h>

```
35142 int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *
35143     restrict attr, int *restrict pshared);
35144 int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
35145     int pshared);
35146
```

35147 **DESCRIPTION**

35148 The *pthread_rwlockattr_getpshared()* function shall obtain the value of the *process-shared* attribute
 35149 from the initialized attributes object referenced by *attr*. The *pthread_rwlockattr_setpshared()*
 35150 function shall set the *process-shared* attribute in an initialized attributes object referenced by *attr*.

35151 The *process-shared* attribute shall be set to PTHREAD_PROCESS_SHARED to permit a read-
 35152 write lock to be operated upon by any thread that has access to the memory where the read-
 35153 write lock is allocated, even if the read-write lock is allocated in memory that is shared by
 35154 multiple processes. If the *process-shared* attribute is PTHREAD_PROCESS_PRIVATE, the read-
 35155 write lock shall only be operated upon by threads created within the same process as the thread
 35156 that initialized the read-write lock; if threads of differing processes attempt to operate on such a
 35157 read-write lock, the behavior is undefined. The default value of the *process-shared* attribute shall
 35158 be PTHREAD_PROCESS_PRIVATE.

35159 Additional attributes, their default values, and the names of the associated functions to get and
 35160 set those attribute values are implementation-defined.

35161 **RETURN VALUE**

35162 Upon successful completion, the *pthread_rwlockattr_getpshared()* function shall return zero and
 35163 store the value of the *process-shared* attribute of *attr* into the object referenced by the *pshared*
 35164 parameter. Otherwise, an error number shall be returned to indicate the error.

35165 If successful, the *pthread_rwlockattr_setpshared()* function shall return zero; otherwise, an error
 35166 number shall be returned to indicate the error.

35167 **ERRORS**

35168 The *pthread_rwlockattr_getpshared()* and *pthread_rwlockattr_setpshared()* functions may fail if:

35169 [EINVAL] The value specified by *attr* is invalid.

35170 The *pthread_rwlockattr_setpshared()* function may fail if:

35171 [EINVAL] The new value specified for the attribute is outside the range of legal values
 35172 for that attribute.

35173 These functions shall not return an error code of [EINTR].

35174 **EXAMPLES**

35175 None.

35176 **APPLICATION USAGE**

35177 None.

35178 **RATIONALE**

35179 None.

35180 **FUTURE DIRECTIONS**

35181 None.

35182 **SEE ALSO**

35183 *pthread_rwlock_destroy()*, *pthread_rwlockattr_destroy()*, *pthread_rwlockattr_init()*, the Base
35184 Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

35185 **CHANGE HISTORY**

35186 First released in Issue 5.

35187 **Issue 6**

35188 The following changes are made for alignment with IEEE Std 1003.1j-2000:

- 35189 • The margin code in the SYNOPSIS is changed to THR TSH to indicate that the functionality
35190 is now part of the Threads option (previously it was part of the Read-Write Locks option in
35191 IEEE Std 1003.1j-2000 and also part of the XSI extension).
- 35192 • The DESCRIPTION notes that additional attributes are implementation-defined.
- 35193 • The SEE ALSO section is updated.

35194 The **restrict** keyword is added to the *pthread_rwlockattr_getpshared()* prototype for alignment
35195 with the ISO/IEC 9899:1999 standard.

35196 **NAME**

35197 pthread_rwlockattr_init — initialize the read-write lock attributes object

35198 **SYNOPSIS**

35199 XSI `#include <pthread.h>`

35200 `int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);`

35201

35202 **DESCRIPTION**

35203 Refer to *pthread_rwlockattr_destroy()*.

35204 NAME

35205 pthread_rwlockattr_setpshared — set the process-shared attribute of the read-write lock
35206 attributes object

35207 SYNOPSIS

35208 XSI `#include <pthread.h>`

35209 `int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,`
35210 `int pshared);`

35211

35212 DESCRIPTION

35213 Refer to *pthread_rwlockattr_getpshared()*.

35214 **NAME**

35215 pthread_self — get the calling thread ID

35216 **SYNOPSIS**

35217 THR #include <pthread.h>

35218 pthread_t pthread_self(void);

35219

35220 **DESCRIPTION**35221 The *pthread_self()* function shall return the thread ID of the calling thread.35222 **RETURN VALUE**

35223 Refer to the DESCRIPTION.

35224 **ERRORS**

35225 No errors are defined.

35226 The *pthread_self()* function shall not return an error code of [EINTR].35227 **EXAMPLES**

35228 None.

35229 **APPLICATION USAGE**

35230 None.

35231 **RATIONALE**

35232 The *pthread_self()* function provides a capability similar to the *getpid()* function for processes
35233 and the rationale is the same: the creation call does not provide the thread ID to the created
35234 thread.

35235 **FUTURE DIRECTIONS**

35236 None.

35237 **SEE ALSO**

35238 *pthread_create()*, *pthread_equal()*, the Base Definitions volume of IEEE Std 1003.1-2001,
35239 <pthread.h>

35240 **CHANGE HISTORY**

35241 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

35242 **Issue 6**35243 The *pthread_self()* function is marked as part of the Threads option.

35244 **NAME**

35245 pthread_setcancelstate, pthread_setcanceltype, pthread_testcancel — set cancelability state

35246 **SYNOPSIS**

35247 THR #include <pthread.h>

35248 int pthread_setcancelstate(int state, int *oldstate);

35249 int pthread_setcanceltype(int type, int *oldtype);

35250 void pthread_testcancel(void);

35251

35252 **DESCRIPTION**

35253 The *pthread_setcancelstate()* function shall atomically both set the calling thread's cancelability state to the indicated *state* and return the previous cancelability state at the location referenced by *oldstate*. Legal values for *state* are PTHREAD_CANCEL_ENABLE and PTHREAD_CANCEL_DISABLE.

35257 The *pthread_setcanceltype()* function shall atomically both set the calling thread's cancelability type to the indicated *type* and return the previous cancelability type at the location referenced by *oldtype*. Legal values for *type* are PTHREAD_CANCEL_DEFERRED and PTHREAD_CANCEL_ASYNCHRONOUS.

35261 The cancelability state and type of any newly created threads, including the thread in which *main()* was first invoked, shall be PTHREAD_CANCEL_ENABLE and PTHREAD_CANCEL_DEFERRED respectively.

35264 The *pthread_testcancel()* function shall create a cancellation point in the calling thread. The *pthread_testcancel()* function shall have no effect if cancelability is disabled.

35266 **RETURN VALUE**

35267 If successful, the *pthread_setcancelstate()* and *pthread_setcanceltype()* functions shall return zero; otherwise, an error number shall be returned to indicate the error.

35269 **ERRORS**35270 The *pthread_setcancelstate()* function may fail if:

35271 [EINVAL] The specified state is not PTHREAD_CANCEL_ENABLE or PTHREAD_CANCEL_DISABLE.

35273 The *pthread_setcanceltype()* function may fail if:

35274 [EINVAL] The specified type is not PTHREAD_CANCEL_DEFERRED or PTHREAD_CANCEL_ASYNCHRONOUS.

35276 These functions shall not return an error code of [EINTR].

35277 **EXAMPLES**

35278 None.

35279 **APPLICATION USAGE**

35280 None.

35281 **RATIONALE**

35282 The *pthread_setcancelstate()* and *pthread_setcanceltype()* functions control the points at which a thread may be asynchronously canceled. For cancellation control to be usable in modular fashion, some rules need to be followed.

35285 An object can be considered to be a generalization of a procedure. It is a set of procedures and global variables written as a unit and called by clients not known by the object. Objects may depend on other objects.

35288 First, cancelability should only be disabled on entry to an object, never explicitly enabled. On
35289 exit from an object, the cancelability state should always be restored to its value on entry to the
35290 object.

35291 This follows from a modularity argument: if the client of an object (or the client of an object that
35292 uses that object) has disabled cancelability, it is because the client does not want to be concerned
35293 about cleaning up if the thread is canceled while executing some sequence of actions. If an object
35294 is called in such a state and it enables cancelability and a cancelation request is pending for that
35295 thread, then the thread is canceled, contrary to the wish of the client that disabled.

35296 Second, the cancelability type may be explicitly set to either *deferred* or *asynchronous* upon entry
35297 to an object. But as with the cancelability state, on exit from an object the cancelability type
35298 should always be restored to its value on entry to the object.

35299 Finally, only functions that are cancel-safe may be called from a thread that is asynchronously
35300 cancelable.

35301 FUTURE DIRECTIONS

35302 None.

35303 SEE ALSO

35304 *pthread_cancel()*, the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

35305 CHANGE HISTORY

35306 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

35307 Issue 6

35308 The *pthread_setcancelstate()*, *pthread_setcanceltype()*, and *pthread_testcancel()* functions are marked
35309 as part of the Threads option.

35310 NAME

35311 pthread_setconcurrency — set the level of concurrency

35312 SYNOPSIS

35313 XSI `#include <pthread.h>`

35314 `int pthread_setconcurrency(int new_level);`

35315

35316 DESCRIPTION

35317 Refer to *pthread_getconcurrency()*.

35318 **NAME**

35319 pthread_setschedparam — dynamic thread scheduling parameters access (**REALTIME**
35320 **THREADS**)

35321 **SYNOPSIS**

35322 THR TPS #include <pthread.h>

35323 int pthread_setschedparam(pthread_t *thread*, int *policy*,
35324 const struct sched_param **param*);

35325

35326 **DESCRIPTION**

35327 Refer to *pthread_getschedparam()*.

35328 **NAME**

35329 pthread_setschedprio — dynamic thread scheduling parameters access (**REALTIME**
 35330 **THREADS**)

35331 **SYNOPSIS**

35332 THR TPS #include <pthread.h>

35333 int pthread_setschedprio(pthread_t thread, int prio);

35334

35335 **DESCRIPTION**

35336 The *pthread_setschedprio()* function shall set the scheduling priority for the thread whose thread
 35337 ID is given by *thread* to the value given by *prio*. See **Scheduling Policies** (on page 44) for a
 35338 description on how this function call affects the ordering of the thread in the thread list for its
 35339 new priority.

35340 If the *pthread_setschedprio()* function fails, the scheduling priority of the target thread shall not be
 35341 changed.

35342 **RETURN VALUE**

35343 If successful, the *pthread_setschedprio()* function shall return zero; otherwise, an error number
 35344 shall be returned to indicate the error.

35345 **ERRORS**

35346 The *pthread_setschedprio()* function may fail if:

35347 [EINVAL] The value of *prio* is invalid for the scheduling policy of the specified thread.

35348 [ENOTSUP] An attempt was made to set the priority to an unsupported value.

35349 [EPERM] The caller does not have the appropriate permission to set the scheduling
 35350 policy of the specified thread.

35351 [EPERM] The implementation does not allow the application to modify the priority to
 35352 the value specified.

35353 [ESRCH] The value specified by *thread* does not refer to an existing thread.

35354 The *pthread_setschedprio()* function shall not return an error code of [EINTR].

35355 **EXAMPLES**

35356 None.

35357 **APPLICATION USAGE**

35358 None.

35359 **RATIONALE**

35360 The *pthread_setschedprio()* function provides a way for an application to temporarily raise its
 35361 priority and then lower it again, without having the undesired side effect of yielding to other
 35362 threads of the same priority. This is necessary if the application is to implement its own
 35363 strategies for bounding priority inversion, such as priority inheritance or priority ceilings. This
 35364 capability is especially important if the implementation does not support the Thread Priority
 35365 Protection or Thread Priority Inheritance options, but even if those options are supported it is
 35366 needed if the application is to bound priority inheritance for other resources, such as
 35367 semaphores.

35368 The standard developers considered that while it might be preferable conceptually to solve this
 35369 problem by modifying the specification of *pthread_setschedparam()*, it was too late to make such a
 35370 change, as there may be implementations that would need to be changed. Therefore, this new
 35371 function was introduced.

35372 **FUTURE DIRECTIONS**

35373 None.

35374 **SEE ALSO**

35375 **Scheduling Policies** (on page 44), *pthread_getschedparam()*, the Base Definitions volume of
35376 IEEE Std 1003.1-2001, <**pthread.h**>

35377 **CHANGE HISTORY**

35378 First released in Issue 6. Included as a response to IEEE PASC Interpretation 1003.1 #96.

35379 NAME

35380 pthread_setspecific — thread-specific data management

35381 SYNOPSIS

35382 THR `#include <pthread.h>`

35383 `int pthread_setspecific(pthread_key_t key, const void *value);`

35384

35385 DESCRIPTION

35386 Refer to *pthread_getspecific()*.

35387 **NAME**

35388 pthread_sigmask, sigprocmask — examine and change blocked signals

35389 **SYNOPSIS**

35390 #include <signal.h>

35391 THR int pthread_sigmask(int how, const sigset_t *restrict set,
35392 sigset_t *restrict oset);35393 CX int sigprocmask(int how, const sigset_t *restrict set,
35394 sigset_t *restrict oset);

35395

35396 **DESCRIPTION**35397 THR The *pthread_sigmask()* function shall examine or change (or both) the calling thread's signal
35398 mask, regardless of the number of threads in the process. The function shall be equivalent to
35399 *sigprocmask()*, without the restriction that the call be made in a single-threaded process.35400 In a single-threaded process, the *sigprocmask()* function shall examine or change (or both) the
35401 signal mask of the calling thread.35402 If the argument *set* is not a null pointer, it points to a set of signals to be used to change the
35403 currently blocked set.35404 The argument *how* indicates the way in which the set is changed, and the application shall
35405 ensure it consists of one of the following values:35406 SIG_BLOCK The resulting set shall be the union of the current set and the signal set
35407 pointed to by *set*.35408 SIG_SETMASK The resulting set shall be the signal set pointed to by *set*.35409 SIG_UNBLOCK The resulting set shall be the intersection of the current set and the
35410 complement of the signal set pointed to by *set*.35411 If the argument *oset* is not a null pointer, the previous mask shall be stored in the location
35412 pointed to by *oset*. If *set* is a null pointer, the value of the argument *how* is not significant and the
35413 process' signal mask shall be unchanged; thus the call can be used to enquire about currently
35414 blocked signals.35415 If there are any pending unblocked signals after the call to *sigprocmask()*, at least one of those
35416 signals shall be delivered before the call to *sigprocmask()* returns.35417 It is not possible to block those signals which cannot be ignored. This shall be enforced by the
35418 system without causing an error to be indicated.35419 If any of the SIGFPE, SIGILL, SIGSEGV, or SIGBUS signals are generated while they are blocked,
35420 the result is undefined, unless the signal was generated by the *kill()* function, the *sigqueue()*
35421 function, or the *raise()* function.35422 If *sigprocmask()* fails, the thread's signal mask shall not be changed.35423 The use of the *sigprocmask()* function is unspecified in a multi-threaded process.35424 **RETURN VALUE**35425 THR Upon successful completion *pthread_sigmask()* shall return 0; otherwise, it shall return the
35426 corresponding error number.35427 Upon successful completion, *sigprocmask()* shall return 0; otherwise, -1 shall be returned, *errno*
35428 shall be set to indicate the error, and the process' signal mask shall be unchanged.

35429 **ERRORS**

35430 THR The *pthread_sigmask()* and *sigprocmask()* functions shall fail if:

35431 [EINVAL] The value of the *how* argument is not equal to one of the defined values.

35432 THR The *pthread_sigmask()* function shall not return an error code of [EINTR].

35433 **EXAMPLES**

35434 None.

35435 **APPLICATION USAGE**

35436 None.

35437 **RATIONALE**

35438 When a process' signal mask is changed in a signal-catching function that is installed by
 35439 *sigaction()*, the restoration of the signal mask on return from the signal-catching function
 35440 overrides that change (see *sigaction()*). If the signal-catching function was installed with
 35441 *signal()*, it is unspecified whether this occurs.

35442 See *kill()* for a discussion of the requirement on delivery of signals.

35443 **FUTURE DIRECTIONS**

35444 None.

35445 **SEE ALSO**

35446 *sigaction()*, *sigaddset()*, *sigdelset()*, *sigemptyset()*, *sigfillset()*, *sigismember()*, *sigpending()*,
 35447 *sigqueue()*, *sigsuspend()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**signal.h**>

35448 **CHANGE HISTORY**

35449 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

35450 **Issue 5**

35451 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

35452 The *pthread_sigmask()* function is added for alignment with the POSIX Threads Extension.

35453 **Issue 6**

35454 The *pthread_sigmask()* function is marked as part of the Threads option.

35455 The SYNOPSIS for *sigprocmask()* is marked as a CX extension to note that the presence of this
 35456 function in the <**signal.h**> header is an extension to the ISO C standard.

35457 The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- 35458 • The DESCRIPTION is updated to explicitly state the functions which may generate the
 35459 signal.

35460 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

35461 The **restrict** keyword is added to the *pthread_sigmask()* and *sigprocmask()* prototypes for
 35462 alignment with the ISO/IEC 9899: 1999 standard.

35463 **NAME**

35464 pthread_spin_destroy, pthread_spin_init — destroy or initialize a spin lock object (ADVANCED
35465 REALTIME THREADS)

35466 **SYNOPSIS**

35467 THR SPI #include <pthread.h>

```
35468 int pthread_spin_destroy(pthread_spinlock_t *lock);
35469 int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
35470
```

35471 **DESCRIPTION**

35472 The *pthread_spin_destroy()* function shall destroy the spin lock referenced by *lock* and release any
35473 resources used by the lock. The effect of subsequent use of the lock is undefined until the lock is
35474 reinitialized by another call to *pthread_spin_init()*. The results are undefined if
35475 *pthread_spin_destroy()* is called when a thread holds the lock, or if this function is called with an
35476 uninitialized thread spin lock.

35477 The *pthread_spin_init()* function shall allocate any resources required to use the spin lock
35478 referenced by *lock* and initialize the lock to an unlocked state.

35479 TSH If the Thread Process-Shared Synchronization option is supported and the value of *pshared* is
35480 PTHREAD_PROCESS_SHARED, the implementation shall permit the spin lock to be operated
35481 upon by any thread that has access to the memory where the spin lock is allocated, even if it is
35482 allocated in memory that is shared by multiple processes.

35483 If the Thread Process-Shared Synchronization option is supported and the value of *pshared* is
35484 PTHREAD_PROCESS_PRIVATE, or if the option is not supported, the spin lock shall only be
35485 operated upon by threads created within the same process as the thread that initialized the spin
35486 lock. If threads of differing processes attempt to operate on such a spin lock, the behavior is
35487 undefined.

35488 The results are undefined if *pthread_spin_init()* is called specifying an already initialized spin
35489 lock. The results are undefined if a spin lock is used without first being initialized.

35490 If the *pthread_spin_init()* function fails, the lock is not initialized and the contents of *lock* are
35491 undefined.

35492 Only the object referenced by *lock* may be used for performing synchronization.

35493 The result of referring to copies of that object in calls to *pthread_spin_destroy()*,
35494 *pthread_spin_lock()*, *pthread_spin_trylock()*, or *pthread_spin_unlock()* is undefined.

35495 **RETURN VALUE**

35496 Upon successful completion, these functions shall return zero; otherwise, an error number shall
35497 be returned to indicate the error.

35498 **ERRORS**

35499 These functions may fail if:

35500 [EBUSY] The implementation has detected an attempt to initialize or destroy a spin
35501 lock while it is in use (for example, while being used in a *pthread_spin_lock()*
35502 call) by another thread.

35503 [EINVAL] The value specified by *lock* is invalid.

35504 The *pthread_spin_init()* function shall fail if:

35505 [EAGAIN] The system lacks the necessary resources to initialize another spin lock.

35506 [ENOMEM] Insufficient memory exists to initialize the lock.

35507 These functions shall not return an error code of [EINTR].

35508 EXAMPLES

35509 None.

35510 APPLICATION USAGE

35511 The *pthread_spin_destroy()* and *pthread_spin_init()* functions are part of the Spin Locks option
35512 and need not be provided on all implementations.

35513 RATIONALE

35514 None.

35515 FUTURE DIRECTIONS

35516 None.

35517 SEE ALSO

35518 *pthread_spin_lock()*, *pthread_spin_unlock()*, the Base Definitions volume of IEEE Std 1003.1-2001,
35519 **<pthread.h>**

35520 CHANGE HISTORY

35521 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

35522 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

35523 **NAME**

35524 pthread_spin_lock, pthread_spin_trylock — lock a spin lock object (**ADVANCED REALTIME**
 35525 **THREADS**)

35526 **SYNOPSIS**

35527 THR SPI #include <pthread.h>

```
35528 int pthread_spin_lock(pthread_spinlock_t *lock);
35529 int pthread_spin_trylock(pthread_spinlock_t *lock);
35530
```

35531 **DESCRIPTION**

35532 The *pthread_spin_lock()* function shall lock the spin lock referenced by *lock*. The calling thread
 35533 shall acquire the lock if it is not held by another thread. Otherwise, the thread shall spin (that is,
 35534 shall not return from the *pthread_spin_lock()* call) until the lock becomes available. The results
 35535 are undefined if the calling thread holds the lock at the time the call is made. The
 35536 *pthread_spin_trylock()* function shall lock the spin lock referenced by *lock* if it is not held by any
 35537 thread. Otherwise, the function shall fail.

35538 The results are undefined if any of these functions is called with an uninitialized spin lock.

35539 **RETURN VALUE**

35540 Upon successful completion, these functions shall return zero; otherwise, an error number shall
 35541 be returned to indicate the error.

35542 **ERRORS**

35543 These functions may fail if:

35544 [EINVAL] The value specified by *lock* does not refer to an initialized spin lock object.

35545 The *pthread_spin_lock()* function may fail if:

35546 [EDEADLK] The calling thread already holds the lock.

35547 The *pthread_spin_trylock()* function shall fail if:

35548 [EBUSY] A thread currently holds the lock.

35549 These functions shall not return an error code of [EINTR].

35550 **EXAMPLES**

35551 None.

35552 **APPLICATION USAGE**

35553 Applications using this function may be subject to priority inversion, as discussed in the Base
 35554 Definitions volume of IEEE Std 1003.1-2001, Section 3.285, Priority Inversion.

35555 The *pthread_spin_lock()* and *pthread_spin_trylock()* functions are part of the Spin Locks option
 35556 and need not be provided on all implementations.

35557 **RATIONALE**

35558 None.

35559 **FUTURE DIRECTIONS**

35560 None.

35561 **SEE ALSO**

35562 *pthread_spin_destroy()*, *pthread_spin_unlock()*, the Base Definitions volume of
 35563 IEEE Std 1003.1-2001, <pthread.h>

35564 CHANGE HISTORY

- 35565 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.
- 35566 In the SYNOPSIS, the inclusion of `<sys/types.h>` is no longer required.

35567 **NAME**35568 pthread_spin_unlock — unlock a spin lock object (**ADVANCED REALTIME THREADS**)35569 **SYNOPSIS**

35570 THR SPI #include <pthread.h>

35571 int pthread_spin_unlock(pthread_spinlock_t *lock);

35572

35573 **DESCRIPTION**

35574 The *pthread_spin_unlock()* function shall release the spin lock referenced by *lock* which was
35575 locked via the *pthread_spin_lock()* or *pthread_spin_trylock()* functions. The results are undefined if
35576 the lock is not held by the calling thread. If there are threads spinning on the lock when
35577 *pthread_spin_unlock()* is called, the lock becomes available and an unspecified spinning thread
35578 shall acquire the lock.

35579 The results are undefined if this function is called with an uninitialized thread spin lock.

35580 **RETURN VALUE**

35581 Upon successful completion, the *pthread_spin_unlock()* function shall return zero; otherwise, an
35582 error number shall be returned to indicate the error.

35583 **ERRORS**

35584 The *pthread_spin_unlock()* function may fail if:

35585 [EINVAL] An invalid argument was specified.

35586 [EPERM] The calling thread does not hold the lock.

35587 This function shall not return an error code of [EINTR].

35588 **EXAMPLES**

35589 None.

35590 **APPLICATION USAGE**

35591 The *pthread_spin_unlock()* function is part of the Spin Locks option and need not be provided on
35592 all implementations.

35593 **RATIONALE**

35594 None.

35595 **FUTURE DIRECTIONS**

35596 None.

35597 **SEE ALSO**

35598 *pthread_spin_destroy()*, *pthread_spin_lock()*, the Base Definitions volume of IEEE Std 1003.1-2001,
35599 <pthread.h>

35600 **CHANGE HISTORY**

35601 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

35602 In the SYNOPSIS, the inclusion of <sys/types.h> is no longer required.

35603 **NAME**

35604 pthread_testcancel — set cancelability state

35605 **SYNOPSIS**

35606 THR `#include <pthread.h>`

35607 `void pthread_testcancel(void);`

35608

35609 **DESCRIPTION**

35610 Refer to *pthread_setcancelstate()*.

35611 **NAME**

35612 ptsname — get name of the slave pseudo-terminal device

35613 **SYNOPSIS**35614 XSI `#include <stdlib.h>`35615 `char *ptsname(int fildes);`

35616

35617 **DESCRIPTION**

35618 The *ptsname()* function shall return the name of the slave pseudo-terminal device associated
35619 with a master pseudo-terminal device. The *fildes* argument is a file descriptor that refers to the
35620 master device. The *ptsname()* function shall return a pointer to a string containing the pathname
35621 of the corresponding slave device.

35622 The *ptsname()* function need not be reentrant. A function that is not required to be reentrant is
35623 not required to be thread-safe.

35624 **RETURN VALUE**

35625 Upon successful completion, *ptsname()* shall return a pointer to a string which is the name of the
35626 pseudo-terminal slave device. Upon failure, *ptsname()* shall return a null pointer. This could
35627 occur if *fildes* is an invalid file descriptor or if the slave device name does not exist in the file
35628 system.

35629 **ERRORS**

35630 No errors are defined.

35631 **EXAMPLES**

35632 None.

35633 **APPLICATION USAGE**35634 The value returned may point to a static data area that is overwritten by each call to *ptsname()*.35635 **RATIONALE**

35636 None.

35637 **FUTURE DIRECTIONS**

35638 None.

35639 **SEE ALSO**

35640 *grantpt()*, *open()*, *ttyname()*, *unlockpt()*, the Base Definitions volume of IEEE Std 1003.1-2001,
35641 `<stdlib.h>`

35642 **CHANGE HISTORY**

35643 First released in Issue 4, Version 2.

35644 **Issue 5**

35645 Moved from X/OPEN UNIX extension to BASE.

35646 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

35647 NAME

35648 `putc` — put a byte on a stream

35649 SYNOPSIS

35650 `#include <stdio.h>`

35651 `int putc(int c, FILE *stream);`

35652 DESCRIPTION

35653 *CX* The functionality described on this reference page is aligned with the ISO C standard. Any
35654 conflict between the requirements described here and the ISO C standard is unintentional. This
35655 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

35656 The `putc()` function shall be equivalent to `fputc()`, except that if it is implemented as a macro it
35657 may evaluate *stream* more than once, so the argument should never be an expression with side
35658 effects.

35659 RETURN VALUE

35660 Refer to `fputc()`.

35661 ERRORS

35662 Refer to `fputc()`.

35663 EXAMPLES

35664 None.

35665 APPLICATION USAGE

35666 Since it may be implemented as a macro, `putc()` may treat a *stream* argument with side effects
35667 incorrectly. In particular, `putc(c,*f++)` does not necessarily work correctly. Therefore, use of this
35668 function is not recommended in such situations; `fputc()` should be used instead.

35669 RATIONALE

35670 None.

35671 FUTURE DIRECTIONS

35672 None.

35673 SEE ALSO

35674 `fputc()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdio.h>`

35675 CHANGE HISTORY

35676 First released in Issue 1. Derived from Issue 1 of the SVID.

35677 **NAME**

35678 putc_unlocked — stdio with explicit client locking

35679 **SYNOPSIS**

35680 TSF #include <stdio.h>

35681 int putc_unlocked(int *c*, FILE **stream*);

35682

35683 **DESCRIPTION**35684 Refer to *getc_unlocked()*.

35685 NAME

35686 `putchar` — put a byte on a stdout stream

35687 SYNOPSIS

35688 `#include <stdio.h>`

35689 `int putchar(int c);`

35690 DESCRIPTION

35691 cx The functionality described on this reference page is aligned with the ISO C standard. Any
35692 conflict between the requirements described here and the ISO C standard is unintentional. This
35693 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

35694 The function call *putchar(c)* shall be equivalent to *putc(c,stdout)*.

35695 RETURN VALUE

35696 Refer to *fputc()*.

35697 ERRORS

35698 Refer to *fputc()*.

35699 EXAMPLES

35700 None.

35701 APPLICATION USAGE

35702 None.

35703 RATIONALE

35704 None.

35705 FUTURE DIRECTIONS

35706 None.

35707 SEE ALSO

35708 *putc()*, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdio.h>`

35709 CHANGE HISTORY

35710 First released in Issue 1. Derived from Issue 1 of the SVID.

35711 **NAME**

35712 putchar_unlocked — stdio with explicit client locking

35713 **SYNOPSIS**35714 TSF `#include <stdio.h>`35715 `int putchar_unlocked(int c);`

35716

35717 **DESCRIPTION**35718 Refer to *getc_unlocked()*.

35719 **NAME**

35720 putenv — change or add a value to an environment

35721 **SYNOPSIS**35722 XSI `#include <stdlib.h>`35723 `int putenv(char *string);`

35724

35725 **DESCRIPTION**

35726 The *putenv()* function shall use the *string* argument to set environment variable values. The
 35727 *string* argument should point to a string of the form "*name=value*". The *putenv()* function shall
 35728 make the value of the environment variable *name* equal to *value* by altering an existing variable
 35729 or creating a new one. In either case, the string pointed to by *string* shall become part of the
 35730 environment, so altering the string shall change the environment. The space used by *string* is no
 35731 longer used once a new string-defining *name* is passed to *putenv()*.

35732 The *putenv()* function need not be reentrant. A function that is not required to be reentrant is not
 35733 required to be thread-safe.

35734 **RETURN VALUE**

35735 Upon successful completion, *putenv()* shall return 0; otherwise, it shall return a non-zero value
 35736 and set *errno* to indicate the error.

35737 **ERRORS**35738 The *putenv()* function may fail if:

35739 [ENOMEM] Insufficient memory was available.

35740 **EXAMPLES**35741 **Changing the Value of an Environment Variable**

35742 The following example changes the value of the *HOME* environment variable to the value
 35743 */usr/home*.

```
35744 #include <stdlib.h>
35745 ...
35746 static char *var = "HOME=/usr/home";
35747 int ret;
35748 ret = putenv(var);
```

35749 **APPLICATION USAGE**

35750 The *putenv()* function manipulates the environment pointed to by *environ*, and can be used in
 35751 conjunction with *getenv()*.

35752 This routine may use *malloc()* to enlarge the environment.

35753 A potential error is to call *putenv()* with an automatic variable as the argument, then return from
 35754 the calling function while *string* is still part of the environment.

35755 The *setenv()* function is preferred over this function.35756 **RATIONALE**

35757 The standard developers noted that *putenv()* is the only function available to add to the
 35758 environment without permitting memory leaks.

35759 **FUTURE DIRECTIONS**

35760 None.

35761 **SEE ALSO**35762 *exec*, *getenv()*, *malloc()*, *setenv()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>35763 **CHANGE HISTORY**

35764 First released in Issue 1. Derived from Issue 1 of the SVID.

35765 **Issue 5**35766 The type of the argument to this function is changed from **const char *** to **char ***. This was
35767 indicated as a FUTURE DIRECTION in previous issues.

35768 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

35769 NAME

35770 putmsg, putpmsg — send a message on a STREAM (STREAMS)

35771 SYNOPSIS

35772 XSR

```
#include <stropts.h>
```

```
35773 int putmsg(int fildes, const struct strbuf *ctlptr,
35774           const struct strbuf *dataptr, int flags);
35775 int putpmsg(int fildes, const struct strbuf *ctlptr,
35776            const struct strbuf *dataptr, int band, int flags);
35777
```

35778 DESCRIPTION

35779 The *putmsg()* function shall create a message from a process buffer(s) and send the message to a
 35780 STREAMS file. The message may contain either a data part, a control part, or both. The data and
 35781 control parts are distinguished by placement in separate buffers, as described below. The
 35782 semantics of each part are defined by the STREAMS module that receives the message.

35783 The *putpmsg()* function is equivalent to *putmsg()*, except that the process can send messages in
 35784 different priority bands. Except where noted, all requirements on *putmsg()* also pertain to
 35785 *putpmsg()*.

35786 The *fildes* argument specifies a file descriptor referencing an open STREAM. The *ctlptr* and
 35787 *dataptr* arguments each point to a **strbuf** structure.

35788 The *ctlptr* argument points to the structure describing the control part, if any, to be included in
 35789 the message. The *buf* member in the **strbuf** structure points to the buffer where the control
 35790 information resides, and the *len* member indicates the number of bytes to be sent. The *maxlen*
 35791 member is not used by *putmsg()*. In a similar manner, the argument *dataptr* specifies the data, if
 35792 any, to be included in the message. The *flags* argument indicates what type of message should be
 35793 sent and is described further below.

35794 To send the data part of a message, the application shall ensure that *dataptr* is not a null pointer
 35795 and the *len* member of *dataptr* is 0 or greater. To send the control part of a message, the
 35796 application shall ensure that the corresponding values are set for *ctlptr*. No data (control) part
 35797 shall be sent if either *dataptr(ctlptr)* is a null pointer or the *len* member of *dataptr(ctlptr)* is set to
 35798 -1.

35799 For *putmsg()*, if a control part is specified and *flags* is set to RS_HIPRI, a high priority message
 35800 shall be sent. If no control part is specified, and *flags* is set to RS_HIPRI, *putmsg()* shall fail and
 35801 set *errno* to [EINVAL]. If *flags* is set to 0, a normal message (priority band equal to 0) shall be
 35802 sent. If a control part and data part are not specified and *flags* is set to 0, no message shall be
 35803 sent and 0 shall be returned.

35804 For *putpmsg()*, the flags are different. The *flags* argument is a bitmask with the following
 35805 mutually-exclusive flags defined: MSG_HIPRI and MSG_BAND. If *flags* is set to 0, *putpmsg()*
 35806 shall fail and set *errno* to [EINVAL]. If a control part is specified and *flags* is set to MSG_HIPRI
 35807 and *band* is set to 0, a high-priority message shall be sent. If *flags* is set to MSG_HIPRI and either
 35808 no control part is specified or *band* is set to a non-zero value, *putpmsg()* shall fail and set *errno* to
 35809 [EINVAL]. If *flags* is set to MSG_BAND, then a message shall be sent in the priority band
 35810 specified by *band*. If a control part and data part are not specified and *flags* is set to MSG_BAND,
 35811 no message shall be sent and 0 shall be returned.

35812 The *putmsg()* function shall block if the STREAM write queue is full due to internal flow control
 35813 conditions, with the following exceptions:

- 35814 • For high-priority messages, *putmsg()* shall not block on this condition and continues
 35815 processing the message.

35816 • For other messages, *putmsg()* shall not block but shall fail when the write queue is full and
 35817 O_NONBLOCK is set.

35818 The *putmsg()* function shall also block, unless prevented by lack of internal resources, while
 35819 waiting for the availability of message blocks in the STREAM, regardless of priority or whether
 35820 O_NONBLOCK has been specified. No partial message shall be sent.

35821 RETURN VALUE

35822 Upon successful completion, *putmsg()* and *putpmsg()* shall return 0; otherwise, they shall return
 35823 −1 and set *errno* to indicate the error.

35824 ERRORS

35825 The *putmsg()* and *putpmsg()* functions shall fail if:

35826 [EAGAIN] A non-priority message was specified, the O_NONBLOCK flag is set, and the
 35827 STREAM write queue is full due to internal flow control conditions; or buffers
 35828 could not be allocated for the message that was to be created.

35829 [EBADF] *fildev* is not a valid file descriptor open for writing.

35830 [EINTR] A signal was caught during *putmsg()*.

35831 [EINVAL] An undefined value is specified in *flags*, or *flags* is set to RS_HIPRI or
 35832 MSG_HIPRI and no control part is supplied, or the STREAM or multiplexer
 35833 referenced by *fildev* is linked (directly or indirectly) downstream from a
 35834 multiplexer, or *flags* is set to MSG_HIPRI and *band* is non-zero (for *putpmsg()*
 35835 only).

35836 [ENOSR] Buffers could not be allocated for the message that was to be created due to
 35837 insufficient STREAMS memory resources.

35838 [ENOSTR] A STREAM is not associated with *fildev*.

35839 [ENXIO] A hangup condition was generated downstream for the specified STREAM.

35840 [EPIPE] or [EIO] The *fildev* argument refers to a STREAMS-based pipe and the other end of the
 35841 pipe is closed. A SIGPIPE signal is generated for the calling thread.

35842 [ERANGE] The size of the data part of the message does not fall within the range
 35843 specified by the maximum and minimum packet sizes of the topmost
 35844 STREAM module. This value is also returned if the control part of the message
 35845 is larger than the maximum configured size of the control part of a message,
 35846 or if the data part of a message is larger than the maximum configured size of
 35847 the data part of a message.

35848 In addition, *putmsg()* and *putpmsg()* shall fail if the STREAM head had processed an
 35849 asynchronous error before the call. In this case, the value of *errno* does not reflect the result of
 35850 *putmsg()* or *putpmsg()*, but reflects the prior error.

35851 **EXAMPLES**35852 **Sending a High-Priority Message**

35853 The value of *fd* is assumed to refer to an open STREAMS file. This call to *putmsg()* does the
 35854 following:

- 35855 1. Creates a high-priority message with a control part and a data part, using the buffers
 35856 pointed to by *ctrlbuf* and *databuf*, respectively.
- 35857 2. Sends the message to the STREAMS file identified by *fd*.

```

35858 #include <stropts.h>
35859 #include <string.h>
35860 ...
35861 int fd;
35862 char *ctrlbuf = "This is the control part";
35863 char *databuf = "This is the data part";
35864 struct strbuf ctrl;
35865 struct strbuf data;
35866 int ret;

35867 ctrl.buf = ctrlbuf;
35868 ctrl.len = strlen(ctrlbuf);

35869 data.buf = databuf;
35870 data.len = strlen(databuf);

35871 ret = putmsg(fd, &ctrl, &data, MSG_HIPRI);

```

35872 **Using putpmsg()**

35873 This example has the same effect as the previous example. In this example, however, the
 35874 *putpmsg()* function creates and sends the message to the STREAMS file.

```

35875 #include <stropts.h>
35876 #include <string.h>
35877 ...
35878 int fd;
35879 char *ctrlbuf = "This is the control part";
35880 char *databuf = "This is the data part";
35881 struct strbuf ctrl;
35882 struct strbuf data;
35883 int ret;

35884 ctrl.buf = ctrlbuf;
35885 ctrl.len = strlen(ctrlbuf);

35886 data.buf = databuf;
35887 data.len = strlen(databuf);

35888 ret = putpmsg(fd, &ctrl, &data, 0, MSG_HIPRI);

```

35889 **APPLICATION USAGE**

35890 None.

35891 **RATIONALE**

35892 None.

35893 **FUTURE DIRECTIONS**

35894 None.

35895 **SEE ALSO**

35896 Section 2.6 (on page 38), *getmsg()*, *poll()*, *read()*, *write()*, the Base Definitions volume of
35897 IEEE Std 1003.1-2001, <**stropts.h**>

35898 **CHANGE HISTORY**

35899 First released in Issue 4, Version 2.

35900 **Issue 5**

35901 Moved from X/OPEN UNIX extension to BASE.

35902 The following text is removed from the DESCRIPTION: “The STREAM head guarantees that the
35903 control part of a message generated by *putmsg()* is at least 64 bytes in length”.

35904 **Issue 6**

35905 This function is marked as part of the XSI STREAMS Option Group.

35906 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

35907 **NAME**

35908 puts — put a string on standard output

35909 **SYNOPSIS**

35910 #include <stdio.h>

35911 int puts(const char *s);

35912 **DESCRIPTION**

35913 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 35914 conflict between the requirements described here and the ISO C standard is unintentional. This
 35915 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

35916 The *puts()* function shall write the string pointed to by *s*, followed by a <newline>, to the
 35917 standard output stream *stdout*. The terminating null byte shall not be written.

35918 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful
 35919 execution of *puts()* and the next successful completion of a call to *fflush()* or *fclose()* on the same
 35920 stream or a call to *exit()* or *abort()*.

35921 **RETURN VALUE**

35922 Upon successful completion, *puts()* shall return a non-negative number. Otherwise, it shall
 35923 CX return EOF, shall set an error indicator for the stream, and *errno* shall be set to indicate the error.

35924 **ERRORS**35925 Refer to *fputc()*.35926 **EXAMPLES**35927 **Printing to Standard Output**

35928 The following example gets the current time, converts it to a string using *localtime()* and
 35929 *asctime()*, and prints it to standard output using *puts()*. It then prints the number of minutes to
 35930 an event for which it is waiting.

```

35931 #include <time.h>
35932 #include <stdio.h>
35933 ...
35934 time_t now;
35935 int minutes_to_event;
35936 ...
35937 time(&now);
35938 printf("The time is ");
35939 puts(asctime(localtime(&now)));
35940 printf("There are %d minutes to the event.\n",
35941        minutes_to_event);
35942 ...

```

35943 **APPLICATION USAGE**35944 The *puts()* function appends a <newline>, while *fputs()* does not.35945 **RATIONALE**

35946 None.

35947 **FUTURE DIRECTIONS**

35948 None.

35949 **SEE ALSO**

35950 *fopen()*, *fputs()*, *putc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

35951 **CHANGE HISTORY**

35952 First released in Issue 1. Derived from Issue 1 of the SVID.

35953 **Issue 6**

35954 Extensions beyond the ISO C standard are marked.

35955 NAME

35956 pututxline — put an entry into the user accounting database

35957 SYNOPSIS

35958 xSI #include <utmpx.h>

35959 struct utmpx *pututxline(const struct utmpx *utmpx);

35960

35961 DESCRIPTION

35962 Refer to *endutxent()*.

35963 **NAME**

35964 putwc — put a wide character on a stream

35965 **SYNOPSIS**

35966 #include <stdio.h>

35967 #include <wchar.h>

35968 wint_t putwc(wchar_t *wc*, FILE **stream*);35969 **DESCRIPTION**

35970 cx The functionality described on this reference page is aligned with the ISO C standard. Any
35971 conflict between the requirements described here and the ISO C standard is unintentional. This
35972 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

35973 The *putwc()* function shall be equivalent to *fputwc()*, except that if it is implemented as a macro
35974 it may evaluate *stream* more than once, so the argument should never be an expression with side
35975 effects.

35976 **RETURN VALUE**35977 Refer to *fputwc()*.35978 **ERRORS**35979 Refer to *fputwc()*.35980 **EXAMPLES**

35981 None.

35982 **APPLICATION USAGE**

35983 Since it may be implemented as a macro, *putwc()* may treat a *stream* argument with side effects
35984 incorrectly. In particular, *putwc(wc,*f++)* need not work correctly. Therefore, use of this function
35985 is not recommended; *fputwc()* should be used instead.

35986 **RATIONALE**

35987 None.

35988 **FUTURE DIRECTIONS**

35989 None.

35990 **SEE ALSO**35991 *fputwc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>, <wchar.h>35992 **CHANGE HISTORY**

35993 First released as a World-wide Portability Interface in Issue 4.

35994 **Issue 5**

35995 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of argument *wc*
35996 is changed from **wint_t** to **wchar_t**.

35997 The Optional Header (OH) marking is removed from <stdio.h>.

35998 **NAME**

35999 putwchar — put a wide character on a stdout stream

36000 **SYNOPSIS**

36001 #include <wchar.h>

36002 wint_t putwchar(wchar_t wc);

36003 **DESCRIPTION**

36004 cx The functionality described on this reference page is aligned with the ISO C standard. Any
36005 conflict between the requirements described here and the ISO C standard is unintentional. This
36006 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

36007 The function call *putwchar(wc)* shall be equivalent to *putwc(wc,stdout)*.

36008 **RETURN VALUE**

36009 Refer to *fputwc()*.

36010 **ERRORS**

36011 Refer to *fputwc()*.

36012 **EXAMPLES**

36013 None.

36014 **APPLICATION USAGE**

36015 None.

36016 **RATIONALE**

36017 None.

36018 **FUTURE DIRECTIONS**

36019 None.

36020 **SEE ALSO**

36021 *fputwc()*, *putwc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

36022 **CHANGE HISTORY**

36023 First released in Issue 4.

36024 **Issue 5**

36025 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of argument *wc*
36026 is changed from **wint_t** to **wchar_t**.

36027 **NAME**

36028 pwrite — write on a file

36029 **SYNOPSIS**

36030 #include <unistd.h>

```
36031 xSI      ssize_t pwrite(int fildes, const void *buf, size_t nbyte,  
36032          off_t offset);
```

36033

36034 **DESCRIPTION**36035 Refer to *write()*.

36036 **NAME**

36037 qsort — sort a table of data

36038 **SYNOPSIS**

36039 #include <stdlib.h>

```
36040       void qsort(void *base, size_t nel, size_t width,  
36041                int (*compar)(const void *, const void *));
```

36042 **DESCRIPTION**

36043 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
36044 conflict between the requirements described here and the ISO C standard is unintentional. This
36045 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

36046 The *qsort()* function shall sort an array of *nel* objects, the initial element of which is pointed to by
36047 *base*. The size of each object, in bytes, is specified by the *width* argument.

36048 The contents of the array shall be sorted in ascending order according to a comparison function.
36049 The *compar* argument is a pointer to the comparison function, which is called with two
36050 arguments that point to the elements being compared. The application shall ensure that the
36051 function returns an integer less than, equal to, or greater than 0, if the first argument is
36052 considered respectively less than, equal to, or greater than the second. If two members compare
36053 as equal, their order in the sorted array is unspecified.

36054 **RETURN VALUE**36055 The *qsort()* function shall not return a value.36056 **ERRORS**

36057 No errors are defined.

36058 **EXAMPLES**

36059 None.

36060 **APPLICATION USAGE**

36061 The comparison function need not compare every byte, so arbitrary data may be contained in
36062 the elements in addition to the values being compared.

36063 **RATIONALE**

36064 None.

36065 **FUTURE DIRECTIONS**

36066 None.

36067 **SEE ALSO**

36068 The Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

36069 **CHANGE HISTORY**

36070 First released in Issue 1. Derived from Issue 1 of the SVID.

36071 **Issue 6**

36072 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

36073 **NAME**

36074 raise — send a signal to the executing process

36075 **SYNOPSIS**

36076 #include <signal.h>

36077 int raise(int *sig*);36078 **DESCRIPTION**

36079 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 36080 conflict between the requirements described here and the ISO C standard is unintentional. This
 36081 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

36082 CX The *raise()* function shall send the signal *sig* to the executing thread or process. If a signal
 36083 handler is called, the *raise()* function shall not return until after the signal handler does.

36084 THR If the implementation supports the Threads option, the effect of the *raise()* function shall be
 36085 equivalent to calling:

36086 pthread_kill(pthread_self(), *sig*);

36087

36088 CX Otherwise, the effect of the *raise()* function shall be equivalent to calling:

36089 kill(getpid(), *sig*);

36090

36091 **RETURN VALUE**

36092 CX Upon successful completion, 0 shall be returned. Otherwise, a non-zero value shall be returned
 36093 and *errno* shall be set to indicate the error.

36094 **ERRORS**36095 The *raise()* function shall fail if:

36096 CX [EINVAL] The value of the *sig* argument is an invalid signal number.

36097 **EXAMPLES**

36098 None.

36099 **APPLICATION USAGE**

36100 None.

36101 **RATIONALE**

36102 The term “thread” is an extension to the ISO C standard.

36103 **FUTURE DIRECTIONS**

36104 None.

36105 **SEE ALSO**

36106 *kill()*, *sigaction()*, the Base Definitions volume of IEEE Std 1003.1-2001, <signal.h>,
 36107 <sys/types.h>

36108 **CHANGE HISTORY**

36109 First released in Issue 4. Derived from the ANSI C standard.

36110 **Issue 5**

36111 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

36112 **Issue 6**

36113 Extensions beyond the ISO C standard are marked.

36114 The following new requirements on POSIX implementations derive from alignment with the
36115 Single UNIX Specification:

- 36116 • In the RETURN VALUE section, the requirement to set *errno* on error is added.
- 36117 • The [EINVAL] error condition is added.

36118 **NAME**

36119 rand, rand_r, srand — pseudo-random number generator

36120 **SYNOPSIS**

36121 #include <stdlib.h>

36122 int rand(void);

36123 TSF int rand_r(unsigned *seed);

36124 void srand(unsigned seed);

36125 **DESCRIPTION**

36126 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 36127 conflict between the requirements described here and the ISO C standard is unintentional. This
 36128 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

36129 The *rand()* function shall compute a sequence of pseudo-random integers in the range
 36130 XSI [0,{RAND_MAX}] with a period of at least 2^{32} .

36131 CX The *rand()* function need not be reentrant. A function that is not required to be reentrant is not
 36132 required to be thread-safe.

36133 TSF The *rand_r()* function shall compute a sequence of pseudo-random integers in the range
 36134 [0,{RAND_MAX}]. (The value of the {RAND_MAX} macro shall be at least 32 767.)

36135 If *rand_r()* is called with the same initial value for the object pointed to by *seed* and that object is
 36136 not modified between successive returns and calls to *rand_r()*, the same sequence shall be
 36137 generated.

36138 The *srand()* function uses the argument as a seed for a new sequence of pseudo-random
 36139 numbers to be returned by subsequent calls to *rand()*. If *srand()* is then called with the same
 36140 seed value, the sequence of pseudo-random numbers shall be repeated. If *rand()* is called before
 36141 any calls to *srand()* are made, the same sequence shall be generated as when *srand()* is first
 36142 called with a seed value of 1.

36143 The implementation shall behave as if no function defined in this volume of
 36144 IEEE Std 1003.1-2001 calls *rand()* or *srand()*.

36145 **RETURN VALUE**36146 The *rand()* function shall return the next pseudo-random number in the sequence.36147 TSF The *rand_r()* function shall return a pseudo-random integer.36148 The *srand()* function shall not return a value.36149 **ERRORS**

36150 No errors are defined.

36151 **EXAMPLES**36152 **Generating a Pseudo-Random Number Sequence**

36153 The following example demonstrates how to generate a sequence of pseudo-random numbers.

36154 #include <stdio.h>

36155 #include <stdlib.h>

36156 ...

36157 long count, i;

36158 char *keyst;

36159 int elementlen, len;

36160 char c;


```

36161     ...
36162     /* Initial random number generator. */
36163     srand(1);

36164     /* Create keys using only lowercase characters */
36165     len = 0;
36166     for (i=0; i<count; i++) {
36167         while (len < elementlen) {
36168             c = (char) (rand() % 128);
36169             if (islower(c))
36170                 keystr[len++] = c;
36171         }

36172         keystr[len] = '\0';
36173         printf("%s Element%0*ld\n", keystr, elementlen, i);
36174         len = 0;
36175     }

```

Generating the Same Sequence on Different Machines

The following code defines a pair of functions that could be incorporated into applications wishing to ensure that the same sequence of numbers is generated across different machines.

```

36179     static unsigned long next = 1;
36180     int myrand(void) /* RAND_MAX assumed to be 32767. */
36181     {
36182         next = next * 1103515245 + 12345;
36183         return((unsigned)(next/65536) % 32768);
36184     }

36185     void mysrand(unsigned seed)
36186     {
36187         next = seed;
36188     }

```

APPLICATION USAGE

The *drand48()* function provides a much more elaborate random number generator.

The limitations on the amount of state that can be carried between one function call and another mean the *rand_r()* function can never be implemented in a way which satisfies all of the requirements on a pseudo-random number generator. Therefore this function should be avoided whenever non-trivial requirements (including safety) have to be fulfilled.

RATIONALE

The ISO C standard *rand()* and *srand()* functions allow per-process pseudo-random streams shared by all threads. Those two functions need not change, but there has to be mutual-exclusion that prevents interference between two threads concurrently accessing the random number generator.

With regard to *rand()*, there are two different behaviors that may be wanted in a multi-threaded program:

1. A single per-process sequence of pseudo-random numbers that is shared by all threads that call *rand()*
2. A different sequence of pseudo-random numbers for each thread that calls *rand()*

36205 This is provided by the modified thread-safe function based on whether the seed value is global
36206 to the entire process or local to each thread.

36207 This does not address the known deficiencies of the *rand()* function implementations, which
36208 have been approached by maintaining more state. In effect, this specifies new thread-safe forms
36209 of a deficient function.

36210 **FUTURE DIRECTIONS**

36211 None.

36212 **SEE ALSO**

36213 *drand48()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

36214 **CHANGE HISTORY**

36215 First released in Issue 1. Derived from Issue 1 of the SVID.

36216 **Issue 5**

36217 The *rand_r()* function is included for alignment with the POSIX Threads Extension.

36218 A note indicating that the *rand()* function need not be reentrant is added to the DESCRIPTION.

36219 **Issue 6**

36220 Extensions beyond the ISO C standard are marked.

36221 The *rand_r()* function is marked as part of the Thread-Safe Functions option.

36222 NAME

36223 random — generate pseudo-random number

36224 SYNOPSIS

36225 xSI `#include <stdlib.h>`

36226 `long random(void);`

36227

36228 DESCRIPTION

36229 Refer to *initstate()*.

36230 **NAME**

36231 pread, read — read from a file

36232 **SYNOPSIS**

36233 #include <unistd.h>

36234 xSI ssize_t pread(int *fildes*, void *buf, size_t *nbyte*, off_t *offset*);36235 ssize_t read(int *fildes*, void *buf, size_t *nbyte*);36236 **DESCRIPTION**

36237 The *read()* function shall attempt to read *nbyte* bytes from the file associated with the open file
 36238 descriptor, *fildes*, into the buffer pointed to by *buf*. The behavior of multiple concurrent reads on
 36239 the same pipe, FIFO, or terminal device is unspecified.

36240 Before any action described below is taken, and if *nbyte* is zero, the *read()* function may detect
 36241 and return errors as described below. In the absence of errors, or if error detection is not
 36242 performed, the *read()* function shall return zero and have no other results.

36243 On files that support seeking (for example, a regular file), the *read()* shall start at a position in
 36244 the file given by the file offset associated with *fildes*. The file offset shall be incremented by the
 36245 number of bytes actually read.

36246 Files that do not support seeking—for example, terminals—always read from the current
 36247 position. The value of a file offset associated with such a file is undefined.

36248 No data transfer shall occur past the current end-of-file. If the starting position is at or after the
 36249 end-of-file, 0 shall be returned. If the file refers to a device special file, the result of subsequent
 36250 *read()* requests is implementation-defined.

36251 If the value of *nbyte* is greater than {SSIZE_MAX}, the result is implementation-defined.

36252 When attempting to read from an empty pipe or FIFO:

- 36253 • If no process has the pipe open for writing, *read()* shall return 0 to indicate end-of-file.
- 36254 • If some process has the pipe open for writing and O_NONBLOCK is set, *read()* shall return
 36255 −1 and set *errno* to [EAGAIN].
- 36256 • If some process has the pipe open for writing and O_NONBLOCK is clear, *read()* shall block
 36257 the calling thread until some data is written or the pipe is closed by all processes that had the
 36258 pipe open for writing.

36259 When attempting to read a file (other than a pipe or FIFO) that supports non-blocking reads and
 36260 has no data currently available:

- 36261 • If O_NONBLOCK is set, *read()* shall return −1 and set *errno* to [EAGAIN].
- 36262 • If O_NONBLOCK is clear, *read()* shall block the calling thread until some data becomes
 36263 available.
- 36264 • The use of the O_NONBLOCK flag has no effect if there is some data available.

36265 The *read()* function reads data previously written to a file. If any portion of a regular file prior to
 36266 the end-of-file has not been written, *read()* shall return bytes with value 0. For example, *lseek()*
 36267 allows the file offset to be set beyond the end of existing data in the file. If data is later written at
 36268 this point, subsequent reads in the gap between the previous end of data and the newly written
 36269 data shall return bytes with value 0 until data is written into the gap.

36270 Upon successful completion, where *nbyte* is greater than 0, *read()* shall mark for update the
 36271 *st_atime* field of the file, and shall return the number of bytes read. This number shall never be
 36272 greater than *nbyte*. The value returned may be less than *nbyte* if the number of bytes left in the

36273	file is less than <i>nbyte</i> , if the <i>read()</i> request was interrupted by a signal, or if the file is a pipe or
36274	FIFO or special file and has fewer than <i>nbyte</i> bytes immediately available for reading. For
36275	example, a <i>read()</i> from a file associated with a terminal may return one typed line of data.
36276	If a <i>read()</i> is interrupted by a signal before it reads any data, it shall return <code>-1</code> with <i>errno</i> set to
36277	<code>[EINTR]</code> .
36278	If a <i>read()</i> is interrupted by a signal after it has successfully read some data, it shall return the
36279	number of bytes read.
36280	For regular files, no data transfer shall occur past the offset maximum established in the open
36281	file description associated with <i>fildes</i> .
36282	If <i>fildes</i> refers to a socket, <i>read()</i> shall be equivalent to <i>recv()</i> with no flags set.
36283 SIO	If the <code>O_DSYNC</code> and <code>O_RSYNC</code> bits have been set, read I/O operations on the file descriptor
36284	shall complete as defined by synchronized I/O data integrity completion. If the <code>O_SYNC</code> and
36285	<code>O_RSYNC</code> bits have been set, read I/O operations on the file descriptor shall complete as
36286	defined by synchronized I/O file integrity completion.
36287 SHM	If <i>fildes</i> refers to a shared memory object, the result of the <i>read()</i> function is unspecified.
36288 TYM	If <i>fildes</i> refers to a typed memory object, the result of the <i>read()</i> function is unspecified.
36289 XSR	A <i>read()</i> from a STREAMS file can read data in three different modes: <i>byte-stream</i> mode,
36290	<i>message-nondiscard</i> mode, and <i>message-discard</i> mode. The default shall be byte-stream mode. This
36291	can be changed using the <code>I_SRDOPT ioctl()</code> request, and can be tested with <code>I_GRDOPT ioctl()</code> .
36292	In byte-stream mode, <i>read()</i> shall retrieve data from the STREAM until as many bytes as were
36293	requested are transferred, or until there is no more data to be retrieved. Byte-stream mode
36294	ignores message boundaries.
36295	In STREAMS message-nondiscard mode, <i>read()</i> shall retrieve data until as many bytes as were
36296	requested are transferred, or until a message boundary is reached. If <i>read()</i> does not retrieve all
36297	the data in a message, the remaining data shall be left on the STREAM, and can be retrieved by
36298	the next <i>read()</i> call. Message-discard mode also retrieves data until as many bytes as were
36299	requested are transferred, or a message boundary is reached. However, unread data remaining
36300	in a message after the <i>read()</i> returns shall be discarded, and shall not be available for a
36301	subsequent <i>read()</i> , <i>getmsg()</i> , or <i>getpmsg()</i> call.
36302	How <i>read()</i> handles zero-byte STREAMS messages is determined by the current read mode
36303	setting. In byte-stream mode, <i>read()</i> shall accept data until it has read <i>nbyte</i> bytes, or until there
36304	is no more data to read, or until a zero-byte message block is encountered. The <i>read()</i> function
36305	shall then return the number of bytes read, and place the zero-byte message back on the
36306	STREAM to be retrieved by the next <i>read()</i> , <i>getmsg()</i> , or <i>getpmsg()</i> . In message-nondiscard mode
36307	or message-discard mode, a zero-byte message shall return 0 and the message shall be removed
36308	from the STREAM. When a zero-byte message is read as the first message on a STREAM, the
36309	message shall be removed from the STREAM and 0 shall be returned, regardless of the read
36310	mode.
36311	A <i>read()</i> from a STREAMS file shall return the data in the message at the front of the STREAM
36312	head read queue, regardless of the priority band of the message.
36313	By default, STREAMs are in control-normal mode, in which a <i>read()</i> from a STREAMS file can
36314	only process messages that contain a data part but do not contain a control part. The <i>read()</i> shall
36315	fail if a message containing a control part is encountered at the STREAM head. This default
36316	action can be changed by placing the STREAM in either control-data mode or control-discard
36317	mode with the <code>I_SRDOPT ioctl()</code> command. In control-data mode, <i>read()</i> shall convert any
36318	control part to data and pass it to the application before passing any data part originally present

36319		in the same message. In control-discard mode, <i>read()</i> shall discard message control parts but
36320		return to the process any data part in the message.
36321		In addition, <i>read()</i> shall fail if the STREAM head had processed an asynchronous error before the
36322		call. In this case, the value of <i>errno</i> shall not reflect the result of <i>read()</i> , but reflect the prior error.
36323		If a hangup occurs on the STREAM being read, <i>read()</i> shall continue to operate normally until
36324		the STREAM head read queue is empty. Thereafter, it shall return 0.
36325 XSI		The <i>pread()</i> function shall be equivalent to <i>read()</i> , except that it shall read from a given position
36326		in the file without changing the file pointer. The first three arguments to <i>pread()</i> are the same as
36327		<i>read()</i> with the addition of a fourth argument <i>offset</i> for the desired position inside the file. An
36328		attempt to perform a <i>pread()</i> on a file that is incapable of seeking shall result in an error.
36329	RETURN VALUE	
36330 XSI		Upon successful completion, <i>read()</i> and <i>pread()</i> shall return a non-negative integer indicating the
36331		number of bytes actually read. Otherwise, the functions shall return -1 and set <i>errno</i> to indicate
36332		the error.
36333	ERRORS	
36334 XSI		The <i>read()</i> and <i>pread()</i> functions shall fail if:
36335	[EAGAIN]	The O_NONBLOCK flag is set for the file descriptor and the process would be
36336		delayed.
36337	[EBADF]	The <i>fildev</i> argument is not a valid file descriptor open for reading.
36338 XSR	[EBADMSG]	The file is a STREAM file that is set to control-normal mode and the message
36339		waiting to be read includes a control part.
36340	[EINTR]	The read operation was terminated due to the receipt of a signal, and no data
36341		was transferred.
36342 XSR	[EINVAL]	The STREAM or multiplexer referenced by <i>fildev</i> is linked (directly or
36343		indirectly) downstream from a multiplexer.
36344	[EIO]	The process is a member of a background process attempting to read from its
36345		controlling terminal, the process is ignoring or blocking the SIGTTIN signal,
36346		or the process group is orphaned. This error may also be generated for
36347		implementation-defined reasons.
36348 XSI	[EISDIR]	The <i>fildev</i> argument refers to a directory and the implementation does not
36349		allow the directory to be read using <i>read()</i> or <i>pread()</i> . The <i>readdir()</i> function
36350		should be used instead.
36351	[EOVERFLOW]	The file is a regular file, <i>nbyte</i> is greater than 0, the starting position is before
36352		the end-of-file, and the starting position is greater than or equal to the offset
36353		maximum established in the open file description associated with <i>fildev</i> .
36354		The <i>read()</i> function shall fail if:
36355	[EAGAIN] or [EWOULDBLOCK]	
36356		The file descriptor is for a socket, is marked O_NONBLOCK, and no data is
36357		waiting to be received.
36358	[ECONNRESET]	A read was attempted on a socket and the connection was forcibly closed by
36359		its peer.
36360	[ENOTCONN]	A read was attempted on a socket that is not connected.
36361	[ETIMEDOUT]	A read was attempted on a socket and a transmission timeout occurred.

36362 XSI The `read()` and `pread()` functions may fail if:

36363 [EIO] A physical I/O error has occurred.

36364 [ENOBUFFS] Insufficient resources were available in the system to perform the operation.

36365 [ENOMEM] Insufficient memory was available to fulfill the request.

36366 [ENXIO] A request was made of a nonexistent device, or the request was outside the

36367 capabilities of the device.

36368 The `pread()` function shall fail, and the file pointer shall remain unchanged, if:

36369 XSI [EINVAL] The *offset* argument is invalid. The value is negative.

36370 XSI [EOVERFLOW] The file is a regular file and an attempt was made to read at or beyond the

36371 offset maximum associated with the file.

36372 XSI [ENXIO] A request was outside the capabilities of the device.

36373 XSI [ESPIPE] *fdes* is associated with a pipe or FIFO.

36374 EXAMPLES

36375 Reading Data into a Buffer

36376 The following example reads data from the file associated with the file descriptor *fd* into the

36377 buffer pointed to by *buf*.

```
36378 #include <sys/types.h>
36379 #include <unistd.h>
36380 ...
36381 char buf[20];
36382 size_t nbytes;
36383 ssize_t bytes_read;
36384 int fd;
36385 ...
36386 nbytes = sizeof(buf);
36387 bytes_read = read(fd, buf, nbytes);
36388 ...
```

36389 APPLICATION USAGE

36390 None.

36391 RATIONALE

36392 This volume of IEEE Std 1003.1-2001 does not specify the value of the file offset after an error is

36393 returned; there are too many cases. For programming errors, such as [EBADF], the concept is

36394 meaningless since no file is involved. For errors that are detected immediately, such as

36395 [EAGAIN], clearly the pointer should not change. After an interrupt or hardware error, however,

36396 an updated value would be very useful and is the behavior of many implementations.

36397 Note that a `read()` of zero bytes does not modify *st_atime*. A `read()` that requests more than zero

36398 bytes, but returns zero, shall modify *st_atime*.

36399 Implementations are allowed, but not required, to perform error checking for `read()` requests of

36400 zero bytes.

Input and Output

The use of I/O with large byte counts has always presented problems. Ideas such as *lread()* and *lwrite()* (using and returning **longs**) were considered at one time. The current solution is to use abstract types on the ISO C standard function to *read()* and *write()*. The abstract types can be declared so that existing functions work, but can also be declared so that larger types can be represented in future implementations. It is presumed that whatever constraints limit the maximum range of **size_t** also limit portable I/O requests to the same range. This volume of IEEE Std 1003.1-2001 also limits the range further by requiring that the byte count be limited so that a signed return value remains meaningful. Since the return type is also a (signed) abstract type, the byte count can be defined by the implementation to be larger than an **int** can hold.

The standard developers considered adding atomicity requirements to a pipe or FIFO, but recognized that due to the nature of pipes and FIFOs there could be no guarantee of atomicity of reads of {PIPE_BUF} or any other size that would be an aid to applications portability.

This volume of IEEE Std 1003.1-2001 requires that no action be taken for *read()* or *write()* when *nbyte* is zero. This is not intended to take precedence over detection of errors (such as invalid buffer pointers or file descriptors). This is consistent with the rest of this volume of IEEE Std 1003.1-2001, but the phrasing here could be misread to require detection of the zero case before any other errors. A value of zero is to be considered a correct value, for which the semantics are a no-op.

I/O is intended to be atomic to ordinary files and pipes and FIFOs. Atomic means that all the bytes from a single operation that started out together end up together, without interleaving from other I/O operations. It is a known attribute of terminals that this is not honored, and terminals are explicitly (and implicitly permanently) excepted, making the behavior unspecified. The behavior for other device types is also left unspecified, but the wording is intended to imply that future standards might choose to specify atomicity (or not).

There were recommendations to add format parameters to *read()* and *write()* in order to handle networked transfers among heterogeneous file system and base hardware types. Such a facility may be required for support by the OSI presentation of layer services. However, it was determined that this should correspond with similar C-language facilities, and that is beyond the scope of this volume of IEEE Std 1003.1-2001. The concept was suggested to the developers of the ISO C standard for their consideration as a possible area for future work.

In 4.3 BSD, a *read()* or *write()* that is interrupted by a signal before transferring any data does not by default return an [EINTR] error, but is restarted. In 4.2 BSD, 4.3 BSD, and the Eighth Edition, there is an additional function, *select()*, whose purpose is to pause until specified activity (data to read, space to write, and so on) is detected on specified file descriptors. It is common in applications written for those systems for *select()* to be used before *read()* in situations (such as keyboard input) where interruption of I/O due to a signal is desired.

The issue of which files or file types are interruptible is considered an implementation design issue. This is often affected primarily by hardware and reliability issues.

There are no references to actions taken following an “unrecoverable error”. It is considered beyond the scope of this volume of IEEE Std 1003.1-2001 to describe what happens in the case of hardware errors.

Previous versions of IEEE Std 1003.1-2001 allowed two very different behaviors with regard to the handling of interrupts. In order to minimize the resulting confusion, it was decided that IEEE Std 1003.1-2001 should support only one of these behaviors. Historical practice on AT&T-derived systems was to have *read()* and *write()* return -1 and set *errno* to [EINTR] when interrupted after some, but not all, of the data requested had been transferred. However, the U.S. Department of Commerce FIPS 151-1 and FIPS 151-2 require the historical BSD behavior, in

which *read()* and *write()* return the number of bytes actually transferred before the interrupt. If *read()* returns *-1* when any data is transferred, it is difficult to recover from the error on a seekable device and impossible on a non-seekable device. Most new implementations support this behavior. The behavior required by IEEE Std 1003.1-2001 is to return the number of bytes transferred.

IEEE Std 1003.1-2001 does not specify when an implementation that buffers *read()*ss actually moves the data into the user-supplied buffer, so an implementation may chose to do this at the latest possible moment. Therefore, an interrupt arriving earlier may not cause *read()* to return a partial byte count, but rather to return *-1* and set *errno* to [EINTR].

Consideration was also given to combining the two previous options, and setting *errno* to [EINTR] while returning a short count. However, not only is there no existing practice that implements this, it is also contradictory to the idea that when *errno* is set, the function responsible shall return *-1*.

36462 FUTURE DIRECTIONS

36463 None.

36464 SEE ALSO

36465 *fcntl()*, *ioctl()*, *lseek()*, *open()*, *pipe()*, *readv()*, the Base Definitions volume of
36466 IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface, `<stropts.h>`, `<sys/uio.h>`,
36467 `<unistd.h>`

36468 CHANGE HISTORY

36469 First released in Issue 1. Derived from Issue 1 of the SVID.

36470 Issue 5

36471 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
36472 Threads Extension.

36473 Large File Summit extensions are added.

36474 The *pread()* function is added.

36475 Issue 6

36476 The DESCRIPTION and ERRORS sections are updated so that references to STREAMS are
36477 marked as part of the XSI STREAMS Option Group.

36478 The following new requirements on POSIX implementations derive from alignment with the
36479 Single UNIX Specification:

- 36480 • The DESCRIPTION now states that if *read()* is interrupted by a signal after it has successfully
36481 read some data, it returns the number of bytes read. In Issue 3, it was optional whether *read()*
36482 returned the number of bytes read, or whether it returned *-1* with *errno* set to [EINTR]. This
36483 is a FIPS requirement.

- 36484 • In the DESCRIPTION, text is added to indicate that for regular files, no data transfer occurs
36485 past the offset maximum established in the open file description associated with *fildev*. This
36486 change is to support large files.

- 36487 • The [EOVERFLOW] mandatory error condition is added.

- 36488 • The [ENXIO] optional error condition is added.

36489 Text referring to sockets is added to the DESCRIPTION.

36490 The following changes were made to align with the IEEE P1003.1a draft standard:

- 36491 • The effect of reading zero bytes is clarified.

36492 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that
36493 *read()* results are unspecified for typed memory objects.

36494 New RATIONALE is added to explain the atomicity requirements for input and output
36495 operations.

36496 The following error conditions are added for operations on sockets: [EAGAIN],
36497 [ECONNRESET], [ENOTCONN], and [ETIMEDOUT].

36498 The [EIO] error is changed to “may fail”.

36499 The following error conditions are added for operations on sockets: [ENOBUFFS] and
36500 [ENOMEM].

36501 The *readv()* function is split out into a separate reference page.

36502 NAME

36503 readdir, readdir_r — read a directory

36504 SYNOPSIS

36505 #include <dirent.h>

36506 struct dirent *readdir(DIR *dirp);

36507 TSF int readdir_r(DIR *restrict dirp, struct dirent *restrict entry,

36508 struct dirent **restrict result);

36509

36510 DESCRIPTION

36511 The type **DIR**, which is defined in the <**dirent.h**> header, represents a *directory stream*, which is
 36512 an ordered sequence of all the directory entries in a particular directory. Directory entries
 36513 represent files; files may be removed from a directory or added to a directory asynchronously to
 36514 the operation of *readdir()*.

36515 The *readdir()* function shall return a pointer to a structure representing the directory entry at the
 36516 current position in the directory stream specified by the argument *dirp*, and position the
 36517 directory stream at the next entry. It shall return a null pointer upon reaching the end of the
 36518 directory stream. The structure **dirent** defined in the <**dirent.h**> header describes a directory
 36519 entry.

36520 The *readdir()* function shall not return directory entries containing empty names. If entries for
 36521 dot or dot-dot exist, one entry shall be returned for dot and one entry shall be returned for dot-
 36522 dot; otherwise, they shall not be returned.

36523 The pointer returned by *readdir()* points to data which may be overwritten by another call to
 36524 *readdir()* on the same directory stream. This data is not overwritten by another call to *readdir()*
 36525 on a different directory stream.

36526 If a file is removed from or added to the directory after the most recent call to *opendir()* or
 36527 *rewinddir()*, whether a subsequent call to *readdir()* returns an entry for that file is unspecified.

36528 The *readdir()* function may buffer several directory entries per actual read operation; *readdir()*
 36529 shall mark for update the *st_atime* field of the directory each time the directory is actually read.

36530 After a call to *fork()*, either the parent or child (but not both) may continue processing the
 36531 XSI directory stream using *readdir()*, *rewinddir()*, or *seekdir()*. If both the parent and child processes
 36532 use these functions, the result is undefined.

36533 If the entry names a symbolic link, the value of the *d_ino* member is unspecified.

36534 The *readdir()* function need not be reentrant. A function that is not required to be reentrant is not
 36535 required to be thread-safe.

36536 TSF The *readdir_r()* function shall initialize the **dirent** structure referenced by *entry* to represent the
 36537 directory entry at the current position in the directory stream referred to by *dirp*, store a pointer
 36538 to this structure at the location referenced by *result*, and position the directory stream at the next
 36539 entry.

36540 The storage pointed to by *entry* shall be large enough for a **dirent** with an array of **char** *d_name*
 36541 members containing at least {NAME_MAX}+1 elements.

36542 Upon successful return, the pointer returned at **result* shall have the same value as the argument
 36543 *entry*. Upon reaching the end of the directory stream, this pointer shall have the value NULL.

36544 The *readdir_r()* function shall not return directory entries containing empty names.

36545 If a file is removed from or added to the directory after the most recent call to *opendir()* or
 36546 *rewinddir()*, whether a subsequent call to *readdir_r()* returns an entry for that file is unspecified.

36547 The *readdir_r()* function may buffer several directory entries per actual read operation; the
 36548 *readdir_r()* function shall mark for update the *st_atime* field of the directory each time the
 36549 directory is actually read.

36550 Applications wishing to check for error situations should set *errno* to 0 before calling *readdir()*. If
 36551 *errno* is set to non-zero on return, an error occurred.

36552 RETURN VALUE

36553 Upon successful completion, *readdir()* shall return a pointer to an object of type **struct dirent**.
 36554 When an error is encountered, a null pointer shall be returned and *errno* shall be set to indicate
 36555 the error. When the end of the directory is encountered, a null pointer shall be returned and *errno*
 36556 is not changed.

36557 TSF If successful, the *readdir_r()* function shall return zero; otherwise, an error number shall be
 36558 returned to indicate the error.

36559 ERRORS

36560 The *readdir()* function shall fail if:

36561 [EOVERFLOW] One of the values in the structure to be returned cannot be represented
 36562 correctly.

36563 The *readdir()* function may fail if:

36564 [EBADF] The *dirp* argument does not refer to an open directory stream.

36565 [ENOENT] The current position of the directory stream is invalid.

36566 The *readdir_r()* function may fail if:

36567 [EBADF] The *dirp* argument does not refer to an open directory stream.

36568 EXAMPLES

36569 The following sample code searches the current directory for the entry *name*.

```
36570 dirp = opendir(".");
36571 while (dirp) {
36572     errno = 0;
36573     if ((dp = readdir(dirp)) != NULL) {
36574         if (strcmp(dp->d_name, name) == 0) {
36575             closedir(dirp);
36576             return FOUND;
36577         }
36578     } else {
36579         if (errno == 0) {
36580             closedir(dirp);
36581             return NOT_FOUND;
36582         }
36583         closedir(dirp);
36584         return READ_ERROR;
36585     }
36586 }
36587 return OPEN_ERROR;
```


36588 **APPLICATION USAGE**

36589 The *readdir()* function should be used in conjunction with *opendir()*, *closedir()*, and *rewinddir()* to
 36590 examine the contents of the directory.

36591 The *readdir_r()* function is thread-safe and shall return values in a user-supplied buffer instead
 36592 of possibly using a static data area that may be overwritten by each call.

36593 **RATIONALE**

36594 The returned value of *readdir()* merely *represents* a directory entry. No equivalence should be
 36595 inferred.

36596 Historical implementations of *readdir()* obtain multiple directory entries on a single read
 36597 operation, which permits subsequent *readdir()* operations to operate from the buffered
 36598 information. Any wording that required each successful *readdir()* operation to mark the
 36599 directory *st_atime* field for update would disallow such historical performance-oriented
 36600 implementations.

36601 Since *readdir()* returns NULL when it detects an error and when the end of the directory is
 36602 encountered, an application that needs to tell the difference must set *errno* to zero before the call
 36603 and check it if NULL is returned. Since the function must not change *errno* in the second case
 36604 and must set it to a non-zero value in the first case, a zero *errno* after a call returning NULL
 36605 indicates end-of-directory; otherwise, an error.

36606 Routines to deal with this problem more directly were proposed:

36607 `int derror (dirp)`

36608 `DIR *dirp;`

36609 `void clearerr (dirp)`

36610 `DIR *dirp;`

36611 The first would indicate whether an error had occurred, and the second would clear the error
 36612 indication. The simpler method involving *errno* was adopted instead by requiring that *readdir()*
 36613 not change *errno* when end-of-directory is encountered.

36614 An error or signal indicating that a directory has changed while open was considered but
 36615 rejected.

36616 The thread-safe version of the directory reading function returns values in a user-supplied buffer
 36617 instead of possibly using a static data area that may be overwritten by each call. Either the
 36618 {NAME_MAX} compile-time constant or the corresponding *pathconf()* option can be used to
 36619 determine the maximum sizes of returned pathnames.

36620 **FUTURE DIRECTIONS**

36621 None.

36622 **SEE ALSO**

36623 *closedir()*, *lstat()*, *opendir()*, *rewinddir()*, *symlink()*, the Base Definitions volume of
 36624 IEEE Std 1003.1-2001, <*dirent.h*>, <*sys/types.h*>

36625 **CHANGE HISTORY**

36626 First released in Issue 2.

36627 **Issue 5**

36628 Large File Summit extensions are added.

36629 The *readdir_r()* function is included for alignment with the POSIX Threads Extension.

36630 A note indicating that the *readdir()* function need not be reentrant is added to the
 36631 DESCRIPTION.

36632 **Issue 6**

36633 The `readdir_r()` function is marked as part of the Thread-Safe Functions option.

36634 The Open Group Corrigendum U026/7 is applied, correcting the prototype for `readdir_r()`.

36635 The Open Group Corrigendum U026/8 is applied, clarifying the wording of the successful
36636 return for the `readdir_r()` function.

36637 The following new requirements on POSIX implementations derive from alignment with the
36638 Single UNIX Specification:

36639 • The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was
36640 required for conforming implementations of previous POSIX specifications, it was not
36641 required for UNIX applications.

36642 • A statement is added to the DESCRIPTION indicating the disposition of certain fields in
36643 **struct dirent** when an entry refers to a symbolic link.

36644 • The [EOVERFLOW] mandatory error condition is added. This change is to support large
36645 files.

36646 • The [ENOENT] optional error condition is added.

36647 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
36648 its avoidance of possibly using a static data area.

36649 The **restrict** keyword is added to the `readdir_r()` prototype for alignment with the
36650 ISO/IEC 9899:1999 standard.

36651 **NAME**

36652 readlink — read the contents of a symbolic link

36653 **SYNOPSIS**

36654 #include <unistd.h>

```
36655     ssize_t readlink(const char *restrict path, char *restrict buf,
36656                     size_t bufsize);
```

36657 **DESCRIPTION**

36658 The *readlink()* function shall place the contents of the symbolic link referred to by *path* in the
 36659 buffer *buf* which has size *bufsize*. If the number of bytes in the symbolic link is less than *bufsize*,
 36660 the contents of the remainder of *buf* are unspecified. If the *buf* argument is not large enough to
 36661 contain the link content, the first *bufsize* bytes shall be placed in *buf*.

36662 If the value of *bufsize* is greater than {SSIZE_MAX}, the result is implementation-defined.

36663 **RETURN VALUE**

36664 Upon successful completion, *readlink()* shall return the count of bytes placed in the buffer.
 36665 Otherwise, it shall return a value of -1, leave the buffer unchanged, and set *errno* to indicate the
 36666 error.

36667 **ERRORS**

36668 The *readlink()* function shall fail if:

36669 [EACCES] Search permission is denied for a component of the path prefix of *path*.

36670 [EINVAL] The *path* argument names a file that is not a symbolic link.

36671 [EIO] An I/O error occurred while reading from the file system.

36672 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 36673 argument.

36674 [ENAMETOOLONG]

36675 The length of the *path* argument exceeds {PATH_MAX} or a pathname
 36676 component is longer than {NAME_MAX}.

36677 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

36678 [ENOTDIR] A component of the path prefix is not a directory.

36679 The *readlink()* function may fail if:

36680 [EACCES] Read permission is denied for the directory.

36681 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 36682 resolution of the *path* argument.

36683 [ENAMETOOLONG]

36684 As a result of encountering a symbolic link in resolution of the *path* argument,
 36685 the length of the substituted pathname string exceeded {PATH_MAX}.

36686 **EXAMPLES**36687 **Reading the Name of a Symbolic Link**

36688 The following example shows how to read the name of a symbolic link named `/modules/pass1`.

```
36689 #include <unistd.h>
36690 char buf[1024];
36691 ssize_t len;
36692 ...
36693 if ((len = readlink("/modules/pass1", buf, sizeof(buf)-1)) != -1)
36694     buf[len] = '\0';
```

36695 **APPLICATION USAGE**

36696 Conforming applications should not assume that the returned contents of the symbolic link are
 36697 null-terminated.

36698 **RATIONALE**

36699 Since IEEE Std 1003.1-2001 does not require any association of file times with symbolic links,
 36700 there is no requirement that file times be updated by *readlink()*. The type associated with *bufsiz*
 36701 is a **size_t** in order to be consistent with both the ISO C standard and the definition of *read()*.
 36702 The behavior specified for *readlink()* when *bufsiz* is zero represents historical practice. For this
 36703 case, the standard developers considered a change whereby *readlink()* would return the number
 36704 of non-null bytes contained in the symbolic link with the buffer *buf* remaining unchanged;
 36705 however, since the **stat** structure member *st_size* value can be used to determine the size of
 36706 buffer necessary to contain the contents of the symbolic link as returned by *readlink()*, this
 36707 proposal was rejected, and the historical practice retained.

36708 **FUTURE DIRECTIONS**

36709 None.

36710 **SEE ALSO**

36711 *lstat()*, *stat()*, *symlink()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<unistd.h>**

36712 **CHANGE HISTORY**

36713 First released in Issue 4, Version 2.

36714 **Issue 5**

36715 Moved from X/OPEN UNIX extension to BASE.

36716 **Issue 6**

36717 The return type is changed to **ssize_t**, to align with the IEEE P1003.1a draft standard.

36718 The following new requirements on POSIX implementations derive from alignment with the
 36719 Single UNIX Specification:

- 36720 • This function is made mandatory.
- 36721 • In this function it is possible for the return value to exceed the range of the type **ssize_t** (since
 36722 **size_t** has a larger range of positive values than **ssize_t**). A sentence restricting the size of
 36723 the **size_t** object is added to the description to resolve this conflict.

36724 The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- 36725 • The FUTURE DIRECTIONS section is changed to None.

36726 The following changes were made to align with the IEEE P1003.1a draft standard:

- 36727 • The [ELOOP] optional error condition is added.

36728 The **restrict** keyword is added to the *readlink()* prototype for alignment with the
36729 ISO/IEC 9899:1999 standard.

36730 **NAME**

36731 readv — read a vector

36732 **SYNOPSIS**36733 XSI `#include <sys/uio.h>`36734 `ssize_t readv(int fildes, const struct iovec *iov, int iovcnt);`

36735

36736 **DESCRIPTION**

36737 The *readv()* function shall be equivalent to *read()*, except as described below. The *readv()*
 36738 function shall place the input data into the *iovcnt* buffers specified by the members of the *iov*
 36739 array: *iov*[0], *iov*[1], ..., *iov*[*iovcnt*−1]. The *iovcnt* argument is valid if greater than 0 and less than
 36740 or equal to {IOV_MAX}.

36741 Each *iovec* entry specifies the base address and length of an area in memory where data should
 36742 be placed. The *readv()* function shall always fill an area completely before proceeding to the
 36743 next.

36744 Upon successful completion, *readv()* shall mark for update the *st_atime* field of the file.

36745 **RETURN VALUE**36746 Refer to *read()*.36747 **ERRORS**36748 Refer to *read()*.36749 In addition, the *readv()* function shall fail if:

36750 [EINVAL] The sum of the *iov_len* values in the *iov* array overflowed an *ssize_t*.

36751 The *readv()* function may fail if:

36752 [EINVAL] The *iovcnt* argument was less than or equal to 0, or greater than {IOV_MAX}.

36753 **EXAMPLES**36754 **Reading Data into an Array**

36755 The following example reads data from the file associated with the file descriptor *fd* into the
 36756 buffers specified by members of the *iov* array.

```

36757 #include <sys/types.h>
36758 #include <sys/uio.h>
36759 #include <unistd.h>
36760 ...
36761 ssize_t bytes_read;
36762 int fd;
36763 char buf0[20];
36764 char buf1[30];
36765 char buf2[40];
36766 int iovcnt;
36767 struct iovec iov[3];

36768 iov[0].iov_base = buf0;
36769 iov[0].iov_len = sizeof(buf0);
36770 iov[1].iov_base = buf1;
36771 iov[1].iov_len = sizeof(buf1);
36772 iov[2].iov_base = buf2;
36773 iov[2].iov_len = sizeof(buf2);

```



```
36774     ...
36775     iovcnt = sizeof(iov) / sizeof(struct iovec);
36776     bytes_read = readv(fd, iov, iovcnt);
36777     ...
```

36778 APPLICATION USAGE

36779 None.

36780 RATIONALE

36781 Refer to *read()*.

36782 FUTURE DIRECTIONS

36783 None.

36784 SEE ALSO

36785 *read()*, *writew()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/uio.h>

36786 CHANGE HISTORY

36787 First released in Issue 4, Version 2.

36788 Issue 6

36789 Split out from the *read()* reference page.

36790 **NAME**

36791 realloc — memory reallocator

36792 **SYNOPSIS**

36793 #include <stdlib.h>

36794 void *realloc(void *ptr, size_t size);

36795 **DESCRIPTION**

36796 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 36797 conflict between the requirements described here and the ISO C standard is unintentional. This
 36798 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

36799 The *realloc()* function shall change the size of the memory object pointed to by *ptr* to the size
 36800 specified by *size*. The contents of the object shall remain unchanged up to the lesser of the new
 36801 and old sizes. If the new size of the memory object would require movement of the object, the
 36802 space for the previous instantiation of the object is freed. If the new size is larger, the contents of
 36803 the newly allocated portion of the object are unspecified. If *size* is 0 and *ptr* is not a null pointer,
 36804 the object pointed to is freed. If the space cannot be allocated, the object shall remain unchanged.

36805 If *ptr* is a null pointer, *realloc()* shall be equivalent to *malloc()* for the specified size.

36806 If *ptr* does not match a pointer returned earlier by *calloc()*, *malloc()*, or *realloc()* or if the space has
 36807 previously been deallocated by a call to *free()* or *realloc()*, the behavior is undefined.

36808 The order and contiguity of storage allocated by successive calls to *realloc()* is unspecified. The
 36809 pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to
 36810 a pointer to any type of object and then used to access such an object in the space allocated (until
 36811 the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object
 36812 disjoint from any other object. The pointer returned shall point to the start (lowest byte address)
 36813 of the allocated space. If the space cannot be allocated, a null pointer shall be returned.

36814 **RETURN VALUE**

36815 Upon successful completion with a size not equal to 0, *realloc()* shall return a pointer to the
 36816 (possibly moved) allocated space. If *size* is 0, either a null pointer or a unique pointer that can be
 36817 successfully passed to *free()* shall be returned. If there is not enough available memory, *realloc()*
 36818 CX shall return a null pointer and set *errno* to [ENOMEM].

36819 **ERRORS**36820 The *realloc()* function shall fail if:

36821 CX [ENOMEM] Insufficient memory is available.

36822 **EXAMPLES**

36823 None.

36824 **APPLICATION USAGE**

36825 None.

36826 **RATIONALE**

36827 None.

36828 **FUTURE DIRECTIONS**

36829 None.

36830 **SEE ALSO**36831 *calloc()*, *free()*, *malloc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

36832 **CHANGE HISTORY**

36833 First released in Issue 1. Derived from Issue 1 of the SVID.

36834 **Issue 6**

36835 Extensions beyond the ISO C standard are marked.

36836 The following new requirements on POSIX implementations derive from alignment with the
36837 Single UNIX Specification:

- 36838 • In the RETURN VALUE section, if there is not enough available memory, the setting of *errno*
36839 to [ENOMEM] is added.
- 36840 • The [ENOMEM] error condition is added.

36841 **NAME**

36842 realpath — resolve a pathname

36843 **SYNOPSIS**

36844 XSI #include <stdlib.h>

```
36845       char *realpath(const char *restrict file_name,
36846                      char *restrict resolved_name);
```

36847

36848 **DESCRIPTION**

36849 The *realpath()* function shall derive, from the pathname pointed to by *file_name*, an absolute
 36850 pathname that names the same file, whose resolution does not involve '.', '..', or symbolic
 36851 links. The generated pathname shall be stored as a null-terminated string, up to a maximum of
 36852 {PATH_MAX} bytes, in the buffer pointed to by *resolved_name*.

36853 **RETURN VALUE**

36854 Upon successful completion, *realpath()* shall return a pointer to the resolved name. Otherwise,
 36855 *realpath()* shall return a null pointer and set *errno* to indicate the error, and the contents of the
 36856 buffer pointed to by *resolved_name* are undefined.

36857 **ERRORS**36858 The *realpath()* function shall fail if:

- | | | |
|-------|----------------|--|
| 36859 | [EACCES] | Read or search permission was denied for a component of <i>file_name</i> . |
| 36860 | [EINVAL] | Either the <i>file_name</i> or <i>resolved_name</i> argument is a null pointer. |
| 36861 | [EIO] | An error occurred while reading from the file system. |
| 36862 | [ELOOP] | A loop exists in symbolic links encountered during resolution of the <i>path</i> |
| 36863 | | argument. |
| 36864 | [ENAMETOOLONG] | |
| 36865 | | The length of the <i>file_name</i> argument exceeds {PATH_MAX} or a pathname |
| 36866 | | component is longer than {NAME_MAX}. |
| 36867 | [ENOENT] | A component of <i>file_name</i> does not name an existing file or <i>file_name</i> points to |
| 36868 | | an empty string. |
| 36869 | [ENOTDIR] | A component of the path prefix is not a directory. |
| 36870 | | The <i>realpath()</i> function may fail if: |
| 36871 | [ELOOP] | More than {SYMLOOP_MAX} symbolic links were encountered during |
| 36872 | | resolution of the <i>path</i> argument. |
| 36873 | [ENAMETOOLONG] | |
| 36874 | | Pathname resolution of a symbolic link produced an intermediate result |
| 36875 | | whose length exceeds {PATH_MAX}. |
| 36876 | [ENOMEM] | Insufficient storage space is available. |

36877 **EXAMPLES**36878 **Generating an Absolute Pathname**

36879 The following example generates an absolute pathname for the file identified by the *symlinkpath*
36880 argument. The generated pathname is stored in the *actualpath* array.

```
36881 #include <stdlib.h>
36882 ...
36883 char *symlinkpath = "/tmp/symlink/file";
36884 char actualpath [PATH_MAX+1];
36885 char *ptr;
36886 ptr = realpath(symlinkpath, actualpath);
```

36887 **APPLICATION USAGE**

36888 None.

36889 **RATIONALE**

36890 None.

36891 **FUTURE DIRECTIONS**

36892 None.

36893 **SEE ALSO**

36894 *getcwd()*, *sysconf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

36895 **CHANGE HISTORY**

36896 First released in Issue 4, Version 2.

36897 **Issue 5**

36898 Moved from X/OPEN UNIX extension to BASE.

36899 **Issue 6**

36900 The **restrict** keyword is added to the *realpath()* prototype for alignment with the
36901 ISO/IEC 9899:1999 standard.

36902 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
36903 [ELOOP] error condition is added.

36904 **NAME**

36905 recv — receive a message from a connected socket

36906 **SYNOPSIS**

36907 #include <sys/socket.h>

36908 ssize_t recv(int *socket*, void **buffer*, size_t *length*, int *flags*);36909 **DESCRIPTION**

36910 The *recv()* function shall receive a message from a connection-mode or connectionless-mode
 36911 socket. It is normally used with connected sockets because it does not permit the application to
 36912 retrieve the source address of received data.

36913 The *recv()* function takes the following arguments:36914 *socket* Specifies the socket file descriptor.36915 *buffer* Points to a buffer where the message should be stored.36916 *length* Specifies the length in bytes of the buffer pointed to by the *buffer* argument.

36917 *flags* Specifies the type of message reception. Values of this argument are formed by
 36918 logically OR'ing zero or more of the following values:

36919 MSG_PEEK Peeks at an incoming message. The data is treated as unread and
 36920 the next *recv()* or similar function shall still return this data.

36921 MSG_OOB Requests out-of-band data. The significance and semantics of
 36922 out-of-band data are protocol-specific.

36923 MSG_WAITALL On SOCK_STREAM sockets this requests that the function block
 36924 until the full amount of data can be returned. The function may
 36925 return the smaller amount of data if the socket is a message-
 36926 based socket, if a signal is caught, if the connection is
 36927 terminated, if MSG_PEEK was specified, or if an error is pending
 36928 for the socket.

36929 The *recv()* function shall return the length of the message written to the buffer pointed to by the
 36930 *buffer* argument. For message-based sockets, such as SOCK_DGRAM and SOCK_SEQPACKET,
 36931 the entire message shall be read in a single operation. If a message is too long to fit in the
 36932 supplied buffer, and MSG_PEEK is not set in the *flags* argument, the excess bytes shall be
 36933 discarded. For stream-based sockets, such as SOCK_STREAM, message boundaries shall be
 36934 ignored. In this case, data shall be returned to the user as soon as it becomes available, and no
 36935 data shall be discarded.

36936 If the MSG_WAITALL flag is not set, data shall be returned only up to the end of the first
 36937 message.

36938 If no messages are available at the socket and O_NONBLOCK is not set on the socket's file
 36939 descriptor, *recv()* shall block until a message arrives. If no messages are available at the socket
 36940 and O_NONBLOCK is set on the socket's file descriptor, *recv()* shall fail and set *errno* to
 36941 [EAGAIN] or [EWOULDBLOCK].

36942 **RETURN VALUE**

36943 Upon successful completion, *recv()* shall return the length of the message in bytes. If no
 36944 messages are available to be received and the peer has performed an orderly shutdown, *recv()*
 36945 shall return 0. Otherwise, -1 shall be returned and *errno* set to indicate the error.

36946 **ERRORS**36947 The *recv()* function shall fail if:

36948 [EAGAIN] or [EWOULDBLOCK]

36949 The socket's file descriptor is marked O_NONBLOCK and no data is waiting
36950 to be received; or MSG_OOB is set and no out-of-band data is available and
36951 either the socket's file descriptor is marked O_NONBLOCK or the socket does
36952 not support blocking to await out-of-band data.

36953 [EBADF] The *socket* argument is not a valid file descriptor.

36954 [ECONNRESET] A connection was forcibly closed by a peer.

36955 [EINTR] The *recv()* function was interrupted by a signal that was caught, before any
36956 data was available.

36957 [EINVAL] The MSG_OOB flag is set and no out-of-band data is available.

36958 [ENOTCONN] A receive is attempted on a connection-mode socket that is not connected.

36959 [ENOTSOCK] The *socket* argument does not refer to a socket.

36960 [EOPNOTSUPP] The specified flags are not supported for this socket type or protocol.

36961 [ETIMEDOUT] The connection timed out during connection establishment, or due to a
36962 transmission timeout on active connection.

36963 The *recv()* function may fail if:

36964 [EIO] An I/O error occurred while reading from or writing to the file system.

36965 [ENOBUFS] Insufficient resources were available in the system to perform the operation.

36966 [ENOMEM] Insufficient memory was available to fulfill the request.

36967 **EXAMPLES**

36968 None.

36969 **APPLICATION USAGE**

36970 The *recv()* function is equivalent to *recvfrom()* with a zero *address_len* argument, and to *read()* if
36971 no flags are used.

36972 The *select()* and *poll()* functions can be used to determine when data is available to be received.36973 **RATIONALE**

36974 None.

36975 **FUTURE DIRECTIONS**

36976 None.

36977 **SEE ALSO**

36978 *poll()*, *read()*, *recvmsg()*, *recvfrom()*, *select()*, *send()*, *sendmsg()*, *sendto()*, *shutdown()*, *socket()*,
36979 *write()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>

36980 **CHANGE HISTORY**

36981 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

36982 **NAME**

36983 recvfrom — receive a message from a socket

36984 **SYNOPSIS**

36985 #include <sys/socket.h>

```
36986       ssize_t recvfrom(int socket, void *restrict buffer, size_t length,
36987                       int flags, struct sockaddr *restrict address,
36988                       socklen_t *restrict address_len);
```

36989 **DESCRIPTION**

36990 The *recvfrom()* function shall receive a message from a connection-mode or connectionless-mode
 36991 socket. It is normally used with connectionless-mode sockets because it permits the application
 36992 to retrieve the source address of received data.

36993 The *recvfrom()* function takes the following arguments:

36994	<i>socket</i>	Specifies the socket file descriptor.
36995	<i>buffer</i>	Points to the buffer where the message should be stored.
36996	<i>length</i>	Specifies the length in bytes of the buffer pointed to by the <i>buffer</i> argument.
36997	<i>flags</i>	Specifies the type of message reception. Values of this argument are formed 36998 by logically OR'ing zero or more of the following values:
36999	MSG_PEEK	Peeks at an incoming message. The data is treated as unread 37000 and the next <i>recvfrom()</i> or similar function shall still return 37001 this data.
37002	MSG_OOB	Requests out-of-band data. The significance and semantics 37003 of out-of-band data are protocol-specific.
37004	MSG_WAITALL	On SOCK_STREAM sockets this requests that the function 37005 block until the full amount of data can be returned. The 37006 function may return the smaller amount of data if the socket 37007 is a message-based socket, if a signal is caught, if the 37008 connection is terminated, if MSG_PEEK was specified, or if 37009 an error is pending for the socket.
37010	<i>address</i>	A null pointer, or points to a sockaddr structure in which the sending address 37011 is to be stored. The length and format of the address depend on the address 37012 family of the socket.
37013	<i>address_len</i>	Specifies the length of the sockaddr structure pointed to by the <i>address</i> 37014 argument.

37015 The *recvfrom()* function shall return the length of the message written to the buffer pointed to by
 37016 RS the *buffer* argument. For message-based sockets, such as SOCK_RAW, SOCK_DGRAM, and
 37017 SOCK_SEQPACKET, the entire message shall be read in a single operation. If a message is too
 37018 long to fit in the supplied buffer, and MSG_PEEK is not set in the *flags* argument, the excess
 37019 bytes shall be discarded. For stream-based sockets, such as SOCK_STREAM, message
 37020 boundaries shall be ignored. In this case, data shall be returned to the user as soon as it becomes
 37021 available, and no data shall be discarded.

37022 If the MSG_WAITALL flag is not set, data shall be returned only up to the end of the first
 37023 message.

37024 Not all protocols provide the source address for messages. If the *address* argument is not a null
 37025 pointer and the protocol provides the source address of messages, the source address of the

37026 received message shall be stored in the **sockaddr** structure pointed to by the *address* argument,
 37027 and the length of this address shall be stored in the object pointed to by the *address_len*
 37028 argument.

37029 If the actual length of the address is greater than the length of the supplied **sockaddr** structure,
 37030 the stored address shall be truncated.

37031 If the *address* argument is not a null pointer and the protocol does not provide the source address
 37032 of messages, the value stored in the object pointed to by *address* is unspecified.

37033 If no messages are available at the socket and O_NONBLOCK is not set on the socket's file
 37034 descriptor, *recvfrom()* shall block until a message arrives. If no messages are available at the
 37035 socket and O_NONBLOCK is set on the socket's file descriptor, *recvfrom()* shall fail and set *errno*
 37036 to [EAGAIN] or [EWOULDBLOCK].

37037 RETURN VALUE

37038 Upon successful completion, *recvfrom()* shall return the length of the message in bytes. If no
 37039 messages are available to be received and the peer has performed an orderly shutdown,
 37040 *recvfrom()* shall return 0. Otherwise, the function shall return -1 and set *errno* to indicate the
 37041 error.

37042 ERRORS

37043 The *recvfrom()* function shall fail if:

37044 [EAGAIN] or [EWOULDBLOCK]

37045 The socket's file descriptor is marked O_NONBLOCK and no data is waiting
 37046 to be received; or MSG_OOB is set and no out-of-band data is available and
 37047 either the socket's file descriptor is marked O_NONBLOCK or the socket does
 37048 not support blocking to await out-of-band data.

37049 [EBADF] The *socket* argument is not a valid file descriptor.

37050 [ECONNRESET] A connection was forcibly closed by a peer.

37051 [EINTR] A signal interrupted *recvfrom()* before any data was available.

37052 [EINVAL] The MSG_OOB flag is set and no out-of-band data is available.

37053 [ENOTCONN] A receive is attempted on a connection-mode socket that is not connected.

37054 [ENOTSOCK] The *socket* argument does not refer to a socket.

37055 [EOPNOTSUPP] The specified flags are not supported for this socket type.

37056 [ETIMEDOUT] The connection timed out during connection establishment, or due to a
 37057 transmission timeout on active connection.

37058 The *recvfrom()* function may fail if:

37059 [EIO] An I/O error occurred while reading from or writing to the file system.

37060 [ENOBUFS] Insufficient resources were available in the system to perform the operation.

37061 [ENOMEM] Insufficient memory was available to fulfill the request.

37062 **EXAMPLES**

37063 None.

37064 **APPLICATION USAGE**37065 The *select()* and *poll()* functions can be used to determine when data is available to be received.37066 **RATIONALE**

37067 None.

37068 **FUTURE DIRECTIONS**

37069 None.

37070 **SEE ALSO**37071 *poll()*, *read()*, *recv()*, *recvmsg()*, *select()*, *send()*, *sendmsg()*, *sendto()*, *shutdown()*, *socket()*, *write()*,37072 the Base Definitions volume of IEEE Std 1003.1-2001, <**sys/socket.h**>37073 **CHANGE HISTORY**

37074 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

37075 NAME

37076 recvmsg — receive a message from a socket

37077 SYNOPSIS

37078 #include <sys/socket.h>

37079 ssize_t recvmsg(int socket, struct msghdr *message, int flags);

37080 DESCRIPTION

37081 The *recvmsg()* function shall receive a message from a connection-mode or connectionless-mode
37082 socket. It is normally used with connectionless-mode sockets because it permits the application
37083 to retrieve the source address of received data.

37084 The *recvmsg()* function takes the following arguments:

37085	<i>socket</i>	Specifies the socket file descriptor.
37086	<i>message</i>	Points to a msghdr structure, containing both the buffer to store the source address and the buffers for the incoming message. The length and format of the address depend on the address family of the socket. The <i>msg_flags</i> member is ignored on input, but may contain meaningful values on output.
37087		
37088		
37089		
37090	<i>flags</i>	Specifies the type of message reception. Values of this argument are formed by logically OR'ing zero or more of the following values:
37091		
37092	MSG_OOB	Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific.
37093		
37094	MSG_PEEK	Peeks at the incoming message.
37095	MSG_WAITALL	On SOCK_STREAM sockets this requests that the function block until the full amount of data can be returned. The function may return the smaller amount of data if the socket is a message-based socket, if a signal is caught, if the connection is terminated, if MSG_PEEK was specified, or if an error is pending for the socket.
37096		
37097		
37098		
37099		
37100		

37101 The *recvmsg()* function shall receive messages from unconnected or connected sockets and shall
37102 return the length of the message.

37103 The *recvmsg()* function shall return the total length of the message. For message-based sockets,
37104 such as SOCK_DGRAM and SOCK_SEQPACKET, the entire message shall be read in a single
37105 operation. If a message is too long to fit in the supplied buffers, and MSG_PEEK is not set in the
37106 *flags* argument, the excess bytes shall be discarded, and MSG_TRUNC shall be set in the
37107 *msg_flags* member of the **msghdr** structure. For stream-based sockets, such as SOCK_STREAM,
37108 message boundaries shall be ignored. In this case, data shall be returned to the user as soon as it
37109 becomes available, and no data shall be discarded.

37110 If the MSG_WAITALL flag is not set, data shall be returned only up to the end of the first
37111 message.

37112 If no messages are available at the socket and O_NONBLOCK is not set on the socket's file
37113 descriptor, *recvmsg()* shall block until a message arrives. If no messages are available at the
37114 socket and O_NONBLOCK is set on the socket's file descriptor, the *recvmsg()* function shall fail
37115 and set *errno* to [EAGAIN] or [EWOULDBLOCK].

37116 In the **msghdr** structure, the *msg_name* and *msg_namelen* members specify the source address if
37117 the socket is unconnected. If the socket is connected, the *msg_name* and *msg_namelen* members
37118 shall be ignored. The *msg_name* member may be a null pointer if no names are desired or
37119 required. The *msg_iov* and *msg_iovlen* fields are used to specify where the received data shall be

37120 stored. *msg_iov* points to an array of **iovec** structures; *msg_iovlen* shall be set to the dimension of
 37121 this array. In each **iovec** structure, the *iov_base* field specifies a storage area and the *iov_len* field
 37122 gives its size in bytes. Each storage area indicated by *msg_iov* is filled with received data in turn
 37123 until all of the received data is stored or all of the areas have been filled.

37124 Upon successful completion, the *msg_flags* member of the message header shall be the bitwise-
 37125 inclusive OR of all of the following flags that indicate conditions detected for the received
 37126 message:

37127 MSG_EOR End-of-record was received (if supported by the protocol).

37128 MSG_OOB Out-of-band data was received.

37129 MSG_TRUNC Normal data was truncated.

37130 MSG_CTRUNC Control data was truncated.

37131 RETURN VALUE

37132 Upon successful completion, *recvmsg()* shall return the length of the message in bytes. If no
 37133 messages are available to be received and the peer has performed an orderly shutdown,
 37134 *recvmsg()* shall return 0. Otherwise, -1 shall be returned and *errno* set to indicate the error.

37135 ERRORS

37136 The *recvmsg()* function shall fail if:

37137 [EAGAIN] or [EWOULDBLOCK]

37138 The socket's file descriptor is marked O_NONBLOCK and no data is waiting
 37139 to be received; or MSG_OOB is set and no out-of-band data is available and
 37140 either the socket's file descriptor is marked O_NONBLOCK or the socket does
 37141 not support blocking to await out-of-band data.

37142 [EBADF] The *socket* argument is not a valid open file descriptor.

37143 [ECONNRESET] A connection was forcibly closed by a peer.

37144 [EINTR] This function was interrupted by a signal before any data was available.

37145 [EINVAL] The sum of the *iov_len* values overflows a **ssize_t**, or the MSG_OOB flag is set
 37146 and no out-of-band data is available.

37147 [EMSGSIZE] The *msg_iovlen* member of the **msghdr** structure pointed to by *message* is less
 37148 than or equal to 0, or is greater than {IOV_MAX}.

37149 [ENOTCONN] A receive is attempted on a connection-mode socket that is not connected.

37150 [ENOTSOCK] The *socket* argument does not refer to a socket.

37151 [EOPNOTSUPP] The specified flags are not supported for this socket type.

37152 [ETIMEDOUT] The connection timed out during connection establishment, or due to a
 37153 transmission timeout on active connection.

37154 The *recvmsg()* function may fail if:

37155 [EIO] An I/O error occurred while reading from or writing to the file system.

37156 [ENOBUFS] Insufficient resources were available in the system to perform the operation.

37157 [ENOMEM] Insufficient memory was available to fulfill the request.

37158 EXAMPLES

37159 None.

37160 APPLICATION USAGE

37161 The *select()* and *poll()* functions can be used to determine when data is available to be received.

37162 RATIONALE

37163 None.

37164 FUTURE DIRECTIONS

37165 None.

37166 SEE ALSO

37167 *poll()*, *recv()*, *recvfrom()*, *select()*, *send()*, *sendmsg()*, *sendto()*, *shutdown()*, *socket()*, the Base

37168 Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>

37169 CHANGE HISTORY

37170 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

37171 NAME

37172 regcomp, regerror, regex, regfree — regular expression matching

37173 SYNOPSIS

37174 #include <regex.h>

37175 int regcomp(regex_t *restrict preg, const char *restrict pattern,
37176 int cflags);37177 size_t regerror(int errcode, const regex_t *restrict preg,
37178 char *restrict errbuf, size_t errbuf_size);37179 int regexexec(const regex_t *restrict preg, const char *restrict string,
37180 size_t nmatch, regmatch_t pmatch[restrict], int eflags);

37181 void regfree(regex_t *preg);

37182 DESCRIPTION

37183 These functions interpret *basic* and *extended* regular expressions as described in the Base
37184 Definitions volume of IEEE Std 1003.1-2001, Chapter 9, Regular Expressions.37185 The **regex_t** structure is defined in <regex.h> and contains at least the following member:

37186

37187

37188

Member Type	Member Name	Description
size_t	re_nsub	Number of parenthesized subexpressions.

37189 The **regmatch_t** structure is defined in <regex.h> and contains at least the following members:

37190

37191

37192

37193

37194

Member Type	Member Name	Description
regoff_t	rm_so	Byte offset from start of <i>string</i> to start of substring.
regoff_t	rm_eo	Byte offset from start of <i>string</i> of the first character after the end of substring.

37195 The *regcomp()* function shall compile the regular expression contained in the string pointed to by
37196 the *pattern* argument and place the results in the structure pointed to by *preg*. The *cflags*
37197 argument is the bitwise-inclusive OR of zero or more of the following flags, which are defined in
37198 the <regex.h> header:

37199 REG_EXTENDED Use Extended Regular Expressions.

37200 REG_ICASE Ignore case in match. (See the Base Definitions volume of
37201 IEEE Std 1003.1-2001, Chapter 9, Regular Expressions.)37202 REG_NOSUB Report only success/fail in *regexexec()*.

37203 REG_NEWLINE Change the handling of <newline>s, as described in the text.

37204 The default regular expression type for *pattern* is a Basic Regular Expression. The application can
37205 specify Extended Regular Expressions using the REG_EXTENDED *cflags* flag.37206 If the REG_NOSUB flag was not set in *cflags*, then *regcomp()* shall set *re_nsub* to the number of
37207 parenthesized subexpressions (delimited by "\(\)" in basic regular expressions or "()" in
37208 extended regular expressions) found in *pattern*.37209 The *regexexec()* function compares the null-terminated string specified by *string* with the compiled
37210 regular expression *preg* initialized by a previous call to *regcomp()*. If it finds a match, *regexexec()*
37211 shall return 0; otherwise, it shall return non-zero indicating either no match or an error. The
37212 *eflags* argument is the bitwise-inclusive OR of zero or more of the following flags, which are
37213 defined in the <regex.h> header:

37214 REG_NOTBOL The first character of the string pointed to by *string* is not the beginning of the
 37215 line. Therefore, the circumflex character ('^'), when taken as a special
 37216 character, shall not match the beginning of *string*.

37217 REG_NOTEOL The last character of the string pointed to by *string* is not the end of the line.
 37218 Therefore, the dollar sign ('\$'), when taken as a special character, shall not
 37219 match the end of *string*.

37220 If *nmatch* is 0 or REG_NOSUB was set in the *cflags* argument to *regcomp()*, then *regexexec()* shall
 37221 ignore the *pmatch* argument. Otherwise, the application shall ensure that the *pmatch* argument
 37222 points to an array with at least *nmatch* elements, and *regexexec()* shall fill in the elements of that
 37223 array with offsets of the substrings of *string* that correspond to the parenthesized subexpressions
 37224 of *pattern*: *pmatch[i].rm_so* shall be the byte offset of the beginning and *pmatch[i].rm_eo* shall be
 37225 one greater than the byte offset of the end of substring *i*. (Subexpression *i* begins at the *i*th
 37226 matched open parenthesis, counting from 1.) Offsets in *pmatch[0]* identify the substring that
 37227 corresponds to the entire regular expression. Unused elements of *pmatch* up to *pmatch[nmatch-1]*
 37228 shall be filled with -1. If there are more than *nmatch* subexpressions in *pattern* (*pattern* itself
 37229 counts as a subexpression), then *regexexec()* shall still do the match, but shall record only the first
 37230 *nmatch* substrings.

37231 When matching a basic or extended regular expression, any given parenthesized subexpression
 37232 of *pattern* might participate in the match of several different substrings of *string*, or it might not
 37233 match any substring even though the pattern as a whole did match. The following rules shall be
 37234 used to determine which substrings to report in *pmatch* when matching regular expressions:

- 37235 1. If subexpression *i* in a regular expression is not contained within another subexpression,
 37236 and it participated in the match several times, then the byte offsets in *pmatch[i]* shall
 37237 delimit the last such match.
- 37238 2. If subexpression *i* is not contained within another subexpression, and it did not participate
 37239 in an otherwise successful match, the byte offsets in *pmatch[i]* shall be -1. A subexpression
 37240 does not participate in the match when:

37241 ' * ' or "\{\}" appears immediately after the subexpression in a basic regular
 37242 expression, or ' * ', ' ? ', or "\{ }" appears immediately after the subexpression in an
 37243 extended regular expression, and the subexpression did not match (matched 0 times)

37244 or:

37245 ' | ' is used in an extended regular expression to select this subexpression or another,
 37246 and the other subexpression matched.

- 37247 3. If subexpression *i* is contained within another subexpression *j*, and *i* is not contained
 37248 within any other subexpression that is contained within *j*, and a match of subexpression *j*
 37249 is reported in *pmatch[j]*, then the match or non-match of subexpression *i* reported in
 37250 *pmatch[i]* shall be as described in 1. and 2. above, but within the substring reported in
 37251 *pmatch[j]* rather than the whole string. The offsets in *pmatch[i]* are still relative to the start
 37252 of *string*.

- 37253 4. If subexpression *i* is contained in subexpression *j*, and the byte offsets in *pmatch[j]* are -1,
 37254 then the pointers in *pmatch[i]* shall also be -1.

- 37255 5. If subexpression *i* matched a zero-length string, then both byte offsets in *pmatch[i]* shall be
 37256 the byte offset of the character or null terminator immediately following the zero-length
 37257 string.

37258 If, when *regexexec()* is called, the locale is different from when the regular expression was
 37259 compiled, the result is undefined.

37260 If REG_NEWLINE is not set in *cflags*, then a <newline> in *pattern* or *string* shall be treated as an
 37261 ordinary character. If REG_NEWLINE is set, then <newline> shall be treated as an ordinary
 37262 character except as follows:

- 37263 1. A <newline> in *string* shall not be matched by a period outside a bracket expression or by
 37264 any form of a non-matching list (see the Base Definitions volume of IEEE Std 1003.1-2001,
 37265 Chapter 9, Regular Expressions).
- 37266 2. A circumflex ('^') in *pattern*, when used to specify expression anchoring (see the Base
 37267 Definitions volume of IEEE Std 1003.1-2001, Section 9.3.8, BRE Expression Anchoring),
 37268 shall match the zero-length string immediately after a <newline> in *string*, regardless of
 37269 the setting of REG_NOTBOL.
- 37270 3. A dollar sign ('\$') in *pattern*, when used to specify expression anchoring, shall match the
 37271 zero-length string immediately before a <newline> in *string*, regardless of the setting of
 37272 REG_NOTEOL.

37273 The *regfree()* function frees any memory allocated by *regcomp()* associated with *preg*.

37274 The following constants are defined as error return values:

37275	REG_NOMATCH	<i>regexexec()</i> failed to match.
37276	REG_BADPAT	Invalid regular expression.
37277	REG_ECOLLATE	Invalid collating element referenced.
37278	REG_ETYPE	Invalid character class type referenced.
37279	REG_EESCAPE	Trailing '\\' in pattern.
37280	REG_ESUBREG	Number in "\digit" invalid or in error.
37281	REG_EBRACK	"[]" imbalance.
37282	REG_EPAREN	"\(\)" or "()" imbalance.
37283	REG_EBRACE	"\{\}" imbalance.
37284	REG_BADBR	Content of "\{\}" invalid: not a number, number too large, more than 37285 two numbers, first larger than second.
37286	REG_ERANGE	Invalid endpoint in range expression.
37287	REG_ESPACE	Out of memory.
37288	REG_BADRPT	'?', '*', or '+' not preceded by valid regular expression.

37289 The *regerror()* function provides a mapping from error codes returned by *regcomp()* and
 37290 *regexexec()* to unspecified printable strings. It generates a string corresponding to the value of the
 37291 *errcode* argument, which the application shall ensure is the last non-zero value returned by
 37292 *regcomp()* or *regexexec()* with the given value of *preg*. If *errcode* is not such a value, the content of
 37293 the generated string is unspecified.

37294 If *preg* is a null pointer, but *errcode* is a value returned by a previous call to *regexexec()* or *regcomp()*,
 37295 the *regerror()* still generates an error string corresponding to the value of *errcode*, but it might not
 37296 be as detailed under some implementations.

37297 If the *errbuf_size* argument is not 0, *regerror()* shall place the generated string into the buffer of
 37298 size *errbuf_size* bytes pointed to by *errbuf*. If the string (including the terminating null) cannot fit
 37299 in the buffer, *regerror()* shall truncate the string and null-terminate the result.

37300 If *errbuf_size* is 0, *regerror()* shall ignore the *errbuf* argument, and return the size of the buffer
 37301 needed to hold the generated string.

37302 If the *preg* argument to *regex()* or *regfree()* is not a compiled regular expression returned by
 37303 *regcomp()*, the result is undefined. A *preg* is no longer treated as a compiled regular expression
 37304 after it is given to *regfree()*.

37305 RETURN VALUE

37306 Upon successful completion, the *regcomp()* function shall return 0. Otherwise, it shall return an
 37307 integer value indicating an error as described in <regex.h>, and the content of *preg* is undefined.
 37308 If a code is returned, the interpretation shall be as given in <regex.h>.

37309 If *regcomp()* detects an invalid RE, it may return REG_BADPAT, or it may return one of the error
 37310 codes that more precisely describes the error.

37311 Upon successful completion, the *regex()* function shall return 0. Otherwise, it shall return
 37312 REG_NOMATCH to indicate no match.

37313 Upon successful completion, the *regerror()* function shall return the number of bytes needed to
 37314 hold the entire generated string, including the null termination. If the return value is greater than
 37315 *errbuf_size*, the string returned in the buffer pointed to by *errbuf* has been truncated.

37316 The *regfree()* function shall not return a value.

37317 ERRORS

37318 No errors are defined.

37319 EXAMPLES

```
37320 #include <regex.h>
37321 /*
37322  * Match string against the extended regular expression in
37323  * pattern, treating errors as no match.
37324  *
37325  * Return 1 for match, 0 for no match.
37326  */
37327 int
37328 match(const char *string, char *pattern)
37329 {
37330     int    status;
37331     regex_t re;
37332     if (regcomp(&re, pattern, REG_EXTENDED|REG_NOSUB) != 0) {
37333         return(0); /* Report error. */
37334     }
37335     status = regexexec(&re, string, (size_t) 0, NULL, 0);
37336     regfree(&re);
37337     if (status != 0) {
37338         return(0); /* Report error. */
37339     }
37340     return(1);
37341 }
```

37342 The following demonstrates how the REG_NOTBOL flag could be used with *regex()* to find all
 37343 substrings in a line that match a pattern supplied by a user. (For simplicity of the example, very
 37344 little error checking is done.)


```

37345 (void) regcomp (&re, pattern, 0);
37346 /* This call to regexec() finds the first match on the line. */
37347 error = regexec (&re, &buffer[0], 1, &pm, 0);
37348 while (error == 0) { /* While matches found. */
37349     /* Substring found between pm.rm_so and pm.rm_eo. */
37350     /* This call to regexec() finds the next match. */
37351     error = regexec (&re, buffer + pm.rm_eo, 1, &pm, REG_NOTBOL);
37352 }

```

37353 APPLICATION USAGE

37354 An application could use:

```

37355 regerror(code, preg, (char *)NULL, (size_t)0)

```

37356 to find out how big a buffer is needed for the generated string, *malloc()* a buffer to hold the
 37357 string, and then call *regerror()* again to get the string. Alternatively, it could allocate a fixed,
 37358 static buffer that is big enough to hold most strings, and then use *malloc()* to allocate a larger
 37359 buffer if it finds that this is too small.

37360 To match a pattern as described in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section
 37361 2.13, Pattern Matching Notation, use the *fnmatch()* function.

37362 RATIONALE

37363 The *regexec()* function must fill in all *nmatch* elements of *pmatch*, where *nmatch* and *pmatch* are
 37364 supplied by the application, even if some elements of *pmatch* do not correspond to
 37365 subexpressions in *pattern*. The application writer should note that there is probably no reason
 37366 for using a value of *nmatch* that is larger than *preg->re_nsub*+1.

37367 The REG_NEWLINE flag supports a use of RE matching that is needed in some applications like
 37368 text editors. In such applications, the user supplies an RE asking the application to find a line
 37369 that matches the given expression. An anchor in such an RE anchors at the beginning or end of
 37370 any line. Such an application can pass a sequence of <newline>-separated lines to *regexec()* as a
 37371 single long string and specify REG_NEWLINE to *regcomp()* to get the desired behavior. The
 37372 application must ensure that there are no explicit <newline>s in *pattern* if it wants to ensure that
 37373 any match occurs entirely within a single line.

37374 The REG_NEWLINE flag affects the behavior of *regexec()*, but it is in the *cflags* parameter to
 37375 *regcomp()* to allow flexibility of implementation. Some implementations will want to generate
 37376 the same compiled RE in *regcomp()* regardless of the setting of REG_NEWLINE and have
 37377 *regexec()* handle anchors differently based on the setting of the flag. Other implementations will
 37378 generate different compiled REs based on the REG_NEWLINE.

37379 The REG_ICASE flag supports the operations taken by the *grep -i* option and the historical
 37380 implementations of *ex* and *vi*. Including this flag will make it easier for application code to be
 37381 written that does the same thing as these utilities.

37382 The substrings reported in *pmatch[]* are defined using offsets from the start of the string rather
 37383 than pointers. Since this is a new interface, there should be no impact on historical
 37384 implementations or applications, and offsets should be just as easy to use as pointers. The
 37385 change to offsets was made to facilitate future extensions in which the string to be searched is
 37386 presented to *regexec()* in blocks, allowing a string to be searched that is not all in memory at
 37387 once.

37388 The type **regoff_t** is used for the elements of *pmatch[]* to ensure that the application can
 37389 represent either the largest possible array in memory (important for an application conforming
 37390 to the Shell and Utilities volume of IEEE Std 1003.1-2001) or the largest possible file (important
 37391 for an application using the extension where a file is searched in chunks).

37392 The standard developers rejected the inclusion of a *regsub()* function that would be used to do
37393 substitutions for a matched RE. While such a routine would be useful to some applications, its
37394 utility would be much more limited than the matching function described here. Both RE parsing
37395 and substitution are possible to implement without support other than that required by the
37396 ISO C standard, but matching is much more complex than substituting. The only difficult part of
37397 substitution, given the information supplied by *regexexec()*, is finding the next character in a string
37398 when there can be multi-byte characters. That is a much larger issue, and one that needs a more
37399 general solution.

37400 The *errno* variable has not been used for error returns to avoid filling the *errno* name space for
37401 this feature.

37402 The interface is defined so that the matched substrings *rm_sp* and *rm_ep* are in a separate
37403 **regmatch_t** structure instead of in **regex_t**. This allows a single compiled RE to be used
37404 simultaneously in several contexts; in *main()* and a signal handler, perhaps, or in multiple
37405 threads of lightweight processes. (The *preg* argument to *regexexec()* is declared with type **const**, so
37406 the implementation is not permitted to use the structure to store intermediate results.) It also
37407 allows an application to request an arbitrary number of substrings from an RE. The number of
37408 subexpressions in the RE is reported in *re_nsub* in *preg*. With this change to *regexexec()*,
37409 consideration was given to dropping the REG_NOSUB flag since the user can now specify this
37410 with a zero *nmatch* argument to *regexexec()*. However, keeping REG_NOSUB allows an
37411 implementation to use a different (perhaps more efficient) algorithm if it knows in *regcomp()*
37412 that no subexpressions need be reported. The implementation is only required to fill in *pmatch* if
37413 *nmatch* is not zero and if REG_NOSUB is not specified. Note that the **size_t** type, as defined in
37414 the ISO C standard, is unsigned, so the description of *regexexec()* does not need to address
37415 negative values of *nmatch*.

37416 REG_NOTBOL was added to allow an application to do repeated searches for the same pattern
37417 in a line. If the pattern contains a circumflex character that should match the beginning of a line,
37418 then the pattern should only match when matched against the beginning of the line. Without
37419 the REG_NOTBOL flag, the application could rewrite the expression for subsequent matches,
37420 but in the general case this would require parsing the expression. The need for REG_NOTEOL is
37421 not as clear; it was added for symmetry.

37422 The addition of the *regerror()* function addresses the historical need for conforming application
37423 programs to have access to error information more than “Function failed to compile/match your
37424 RE for unknown reasons”.

37425 This interface provides for two different methods of dealing with error conditions. The specific
37426 error codes (REG_EBRACE, for example), defined in **<regex.h>**, allow an application to recover
37427 from an error if it is so able. Many applications, especially those that use patterns supplied by a
37428 user, will not try to deal with specific error cases, but will just use *regerror()* to obtain a human-
37429 readable error message to present to the user.

37430 The *regerror()* function uses a scheme similar to *confstr()* to deal with the problem of allocating
37431 memory to hold the generated string. The scheme used by *strerror()* in the ISO C standard was
37432 considered unacceptable since it creates difficulties for multi-threaded applications.

37433 The *preg* argument is provided to *regerror()* to allow an implementation to generate a more
37434 descriptive message than would be possible with *errcode* alone. An implementation might, for
37435 example, save the character offset of the offending character of the pattern in a field of *preg*, and
37436 then include that in the generated message string. The implementation may also ignore *preg*.

37437 A REG_FILENAME flag was considered, but omitted. This flag caused *regexexec()* to match
37438 patterns as described in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.13,
37439 Pattern Matching Notation instead of REs. This service is now provided by the *fnmatch()*

37440 function.

37441 Notice that there is a difference in philosophy between the ISO POSIX-2:1993 standard and
37442 IEEE Std 1003.1-2001 in how to handle a “bad” regular expression. The ISO POSIX-2:1993
37443 standard says that many bad constructs “produce undefined results”, or that “the interpretation
37444 is undefined”. IEEE Std 1003.1-2001, however, says that the interpretation of such REs is
37445 unspecified. The term “undefined” means that the action by the application is an error, of
37446 similar severity to passing a bad pointer to a function.

37447 The *regcomp()* and *regexexec()* functions are required to accept any null-terminated string as the
37448 *pattern* argument. If the meaning of the string is “undefined”, the behavior of the function is
37449 “unspecified”. IEEE Std 1003.1-2001 does not specify how the functions will interpret the
37450 pattern; they might return error codes, or they might do pattern matching in some completely
37451 unexpected way, but they should not do something like abort the process.

37452 FUTURE DIRECTIONS

37453 None.

37454 SEE ALSO

37455 *fnmatch()*, *glob()*, Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.13, Pattern
37456 Matching Notation, Base Definitions volume of IEEE Std 1003.1-2001, Chapter 9, Regular
37457 Expressions, **<regex.h>**, **<sys/types.h>**

37458 CHANGE HISTORY

37459 First released in Issue 4. Derived from the ISO POSIX-2 standard.

37460 Issue 5

37461 Moved from POSIX2 C-language Binding to BASE.

37462 Issue 6

37463 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

37464 The following new requirements on POSIX implementations derive from alignment with the
37465 Single UNIX Specification:

- 37466 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
37467 required for conforming implementations of previous POSIX specifications, it was not
37468 required for UNIX applications.

37469 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

37470 The REG_ENOSYS constant is removed.

37471 The **restrict** keyword is added to the *regcomp()*, *regerror()*, and *regexexec()* prototypes for
37472 alignment with the ISO/IEC 9899:1999 standard.

37473 **NAME**

37474 remainder, remainderf, remainderl — remainder function

37475 **SYNOPSIS**

37476 #include <math.h>

37477 double remainder(double x, double y);

37478 float remainderf(float x, float y);

37479 long double remainderl(long double x, long double y);

37480 **DESCRIPTION**

37481 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 37482 conflict between the requirements described here and the ISO C standard is unintentional. This
 37483 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

37484 These functions shall return the floating-point remainder $r = x - ny$ when y is non-zero. The value
 37485 n is the integral value nearest the exact value x/y . When $|n - x/y| = 1/2$, the value n is chosen to
 37486 be even.

37487 The behavior of *remainder()* shall be independent of the rounding mode.37488 **RETURN VALUE**

37489 Upon successful completion, these functions shall return the floating-point remainder $r = x - ny$
 37490 when y is non-zero.

37491 MX If x or y is NaN, a NaN shall be returned.

37492 If x is infinite or y is 0 and the other is non-NaN, a domain error shall occur, and either a NaN (if
 37493 supported), or an implementation-defined value shall be returned.

37494 **ERRORS**

37495 These functions shall fail if:

37496 MX	Domain Error	The x argument is $\pm\text{Inf}$, or the y argument is ± 0 and the other argument is non-NaN.
----------	--------------	---

37498	37499	37500	37501	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then <i>errno</i> shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.
-------	-------	-------	-------	--

37502 **EXAMPLES**

37503 None.

37504 **APPLICATION USAGE**

37505 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
 37506 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

37507 **RATIONALE**

37508 None.

37509 **FUTURE DIRECTIONS**

37510 None.

37511 **SEE ALSO**

37512 *abs()*, *div()*, *feclearexcept()*, *fetestexcept()*, *ldiv()*, the Base Definitions volume of
 37513 IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
 37514 <math.h>

37515 **CHANGE HISTORY**

37516 First released in Issue 4, Version 2.

37517 **Issue 5**

37518 Moved from X/OPEN UNIX extension to BASE.

37519 **Issue 6**

37520 The *remainder()* function is no longer marked as an extension.

37521 The *remainderf()* and *remainderl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

37523 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

37525 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

37527 **NAME**

37528 remove — remove a file

37529 **SYNOPSIS**

37530 #include <stdio.h>

37531 int remove(const char *path);

37532 **DESCRIPTION**

37533 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 37534 conflict between the requirements described here and the ISO C standard is unintentional. This
 37535 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

37536 The *remove()* function shall cause the file named by the pathname pointed to by *path* to be no
 37537 longer accessible by that name. A subsequent attempt to open that file using that name shall fail,
 37538 unless it is created anew.

37539 CX If *path* does not name a directory, *remove(path)* shall be equivalent to *unlink(path)*.

37540 If *path* names a directory, *remove(path)* shall be equivalent to *rmdir(path)*.

37541 **RETURN VALUE**

37542 CX Refer to *rmdir()* or *unlink()*.

37543 **ERRORS**

37544 CX Refer to *rmdir()* or *unlink()*.

37545 **EXAMPLES**37546 **Removing Access to a File**

37547 The following example shows how to remove access to a file named */home/cnd/old_mods*.

```
37548 #include <stdio.h>
37549 int status;
37550 ...
37551 status = remove("/home/cnd/old_mods");
```

37552 **APPLICATION USAGE**

37553 None.

37554 **RATIONALE**

37555 None.

37556 **FUTURE DIRECTIONS**

37557 None.

37558 **SEE ALSO**

37559 *rmdir()*, *unlink()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

37560 **CHANGE HISTORY**

37561 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard and the ISO C
 37562 standard.

37563 **Issue 6**

37564 Extensions beyond the ISO C standard are marked.

37565 The following new requirements on POSIX implementations derive from alignment with the
 37566 Single UNIX Specification:

37567
37568
37569

- The DESCRIPTION, RETURN VALUE, and ERRORS sections are updated so that if *path* is not a directory, *remove()* is equivalent to *unlink()*, and if it is a directory, it is equivalent to *rmdir()*.

37570 NAME

37571 remque — remove an element from a queue

37572 SYNOPSIS

37573 xSI #include <search.h>

37574 void remque(void **element*);

37575

37576 DESCRIPTION

37577 Refer to *insque()*.

37578 **NAME**

37579 remquo, remquof, remquol — remainder functions

37580 **SYNOPSIS**

37581 #include <math.h>

37582 double remquo(double x, double y, int *quo);

37583 float remquof(float x, float y, int *quo);

37584 long double remquol(long double x, long double y, int *quo);

37585 **DESCRIPTION**

37586 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 37587 conflict between the requirements described here and the ISO C standard is unintentional. This
 37588 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

37589 The *remquo()*, *remquof()*, and *remquol()* functions shall compute the same remainder as the
 37590 *remainder()*, *remainderf()*, and *remainderl()* functions, respectively. In the object pointed to by
 37591 *quo*, they store a value whose sign is the sign of x/y and whose magnitude is congruent modulo
 37592 2^n to the magnitude of the integral quotient of x/y , where n is an implementation-defined
 37593 integer greater than or equal to 3.

37594 An application wishing to check for error situations should set *errno* to zero and call
 37595 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 37596 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 37597 zero, an error has occurred.

37598 **RETURN VALUE**37599 These functions shall return $x \text{ REM } y$.37600 MX If x or y is NaN, a NaN shall be returned.

37601 If x is $\pm\text{Inf}$ or y is zero and the other argument is non-NaN, a domain error shall occur, and either
 37602 a NaN (if supported), or an implementation-defined value shall be returned.

37603 **ERRORS**

37604 These functions shall fail if:

37605 MX	Domain Error	The x argument is $\pm\text{Inf}$, or the y argument is ± 0 and the other argument is non-NaN.
----------	--------------	---

37607	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then <i>errno</i> shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.
37608	
37609	
37610	

37611 **EXAMPLES**

37612 None.

37613 **APPLICATION USAGE**

37614 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
 37615 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

37616 **RATIONALE**

37617 These functions are intended for implementing argument reductions which can exploit a few
 37618 low-order bits of the quotient. Note that x may be so large in magnitude relative to y that an
 37619 exact representation of the quotient is not practical.

37620 **FUTURE DIRECTIONS**

37621 None.

37622 **SEE ALSO**

37623 *feclearexcept()*, *fetetestexcept()*, *remainder()*, the Base Definitions volume of IEEE Std 1003.1-2001,
37624 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

37625 **CHANGE HISTORY**

37626 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

37627 **NAME**

37628 rename — rename a file

37629 **SYNOPSIS**

37630 #include <stdio.h>

37631 int rename(const char *old, const char *new);

37632 **DESCRIPTION**

37633 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 37634 conflict between the requirements described here and the ISO C standard is unintentional. This
 37635 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

37636 The *rename()* function shall change the name of a file. The *old* argument points to the pathname
 37637 of the file to be renamed. The *new* argument points to the new pathname of the file.

37638 CX If either the *old* or *new* argument names a symbolic link, *rename()* shall operate on the symbolic
 37639 link itself, and shall not resolve the last component of the argument. If the *old* argument and the
 37640 *new* argument resolve to the same existing file, *rename()* shall return successfully and perform no
 37641 other action.

37642 If the *old* argument points to the pathname of a file that is not a directory, the *new* argument shall
 37643 not point to the pathname of a directory. If the link named by the *new* argument exists, it shall be
 37644 removed and *old* renamed to *new*. In this case, a link named *new* shall remain visible to other
 37645 processes throughout the renaming operation and refer either to the file referred to by *new* or *old*
 37646 before the operation began. Write access permission is required for both the directory containing
 37647 *old* and the directory containing *new*.

37648 If the *old* argument points to the pathname of a directory, the *new* argument shall not point to the
 37649 pathname of a file that is not a directory. If the directory named by the *new* argument exists, it
 37650 shall be removed and *old* renamed to *new*. In this case, a link named *new* shall exist throughout
 37651 the renaming operation and shall refer either to the directory referred to by *new* or *old* before the
 37652 operation began. If *new* names an existing directory, it shall be required to be an empty directory.

37653 If the *old* argument points to a pathname of a symbolic link, the symbolic link shall be renamed.
 37654 If the *new* argument points to a pathname of a symbolic link, the symbolic link shall be removed.

37655 The *new* pathname shall not contain a path prefix that names *old*. Write access permission is
 37656 required for the directory containing *old* and the directory containing *new*. If the *old* argument
 37657 points to the pathname of a directory, write access permission may be required for the directory
 37658 named by *old*, and, if it exists, the directory named by *new*.

37659 If the link named by the *new* argument exists and the file's link count becomes 0 when it is
 37660 removed and no process has the file open, the space occupied by the file shall be freed and the
 37661 file shall no longer be accessible. If one or more processes have the file open when the last link is
 37662 removed, the link shall be removed before *rename()* returns, but the removal of the file contents
 37663 shall be postponed until all references to the file are closed.

37664 Upon successful completion, *rename()* shall mark for update the *st_ctime* and *st_mtime* fields of
 37665 the parent directory of each file.

37666 If the *rename()* function fails for any reason other than [EIO], any file named by *new* shall be
 37667 unaffected.

37668 **RETURN VALUE**

37669 CX Upon successful completion, *rename()* shall return 0; otherwise, -1 shall be returned, *errno* shall
 37670 be set to indicate the error, and neither the file named by *old* nor the file named by *new* shall be
 37671 changed or created.

37672 **ERRORS**37673 The *rename()* function shall fail if:

37674 CX 37675 37676 37677	[EACCES]	A component of either path prefix denies search permission; or one of the directories containing <i>old</i> or <i>new</i> denies write permissions; or, write permission is required and is denied for a directory pointed to by the <i>old</i> or <i>new</i> arguments.
37678 CX 37679	[EBUSY]	The directory named by <i>old</i> or <i>new</i> is currently in use by the system or another process, and the implementation considers this an error.
37680 CX 37681	[EEXIST] or [ENOTEMPTY]	The link named by <i>new</i> is a directory that is not an empty directory.
37682 CX 37683	[EINVAL]	The <i>new</i> directory pathname contains a path prefix that names the <i>old</i> directory.
37684 CX	[EIO]	A physical I/O error has occurred.
37685 CX 37686	[EISDIR]	The <i>new</i> argument points to a directory and the <i>old</i> argument points to a file that is not a directory.
37687 CX 37688	[ELOOP]	A loop exists in symbolic links encountered during resolution of the <i>path</i> argument.
37689 CX 37690	[EMLINK]	The file named by <i>old</i> is a directory, and the link count of the parent directory of <i>new</i> would exceed {LINK_MAX}.
37691 CX 37692 37693	[ENAMETOOLONG]	The length of the <i>old</i> or <i>new</i> argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
37694 CX 37695	[ENOENT]	The link named by <i>old</i> does not name an existing file, or either <i>old</i> or <i>new</i> points to an empty string.
37696 CX	[ENOSPC]	The directory that would contain <i>new</i> cannot be extended.
37697 CX 37698	[ENOTDIR]	A component of either path prefix is not a directory; or the <i>old</i> argument names a directory and <i>new</i> argument names a non-directory file.
37699 XSI 37700 37701 37702 37703 37704 37705	[EPERM] or [EACCES]	The S_ISVTX flag is set on the directory containing the file referred to by <i>old</i> and the caller is not the file owner, nor is the caller the directory owner, nor does the caller have appropriate privileges; or <i>new</i> refers to an existing file, the S_ISVTX flag is set on the directory containing this file, and the caller is not the file owner, nor is the caller the directory owner, nor does the caller have appropriate privileges.
37706 CX 37707	[EROFS]	The requested operation requires writing in a directory on a read-only file system.
37708 CX 37709	[EXDEV]	The links named by <i>new</i> and <i>old</i> are on different file systems and the implementation does not support links between file systems.
37710	The <i>rename()</i> function may fail if:	
37711 XSI	[EBUSY]	The file named by the <i>old</i> or <i>new</i> arguments is a named STREAM.
37712 CX 37713	[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the <i>path</i> argument.

37714 CX [ENAMETOOLONG]
 37715 As a result of encountering a symbolic link in resolution of the *path* argument,
 37716 the length of the substituted pathname string exceeded {PATH_MAX}.

37717 CX [ETXTBSY] The file to be renamed is a pure procedure (shared text) file that is being
 37718 executed.

37719 EXAMPLES**37720 Renaming a File**

37721 The following example shows how to rename a file named `/home/cnd/mod1` to
 37722 `/home/cnd/mod2`.

```
37723 #include <stdio.h>
37724 int status;
37725 ...
37726 status = rename("/home/cnd/mod1", "/home/cnd/mod2");
```

37727 APPLICATION USAGE

37728 Some implementations mark for update the *st_ctime* field of renamed files and some do not.
 37729 Applications which make use of the *st_ctime* field may behave differently with respect to
 37730 renamed files unless they are designed to allow for either behavior.

37731 RATIONALE

37732 This *rename()* function is equivalent for regular files to that defined by the ISO C standard. Its
 37733 inclusion here expands that definition to include actions on directories and specifies behavior
 37734 when the *new* parameter names a file that already exists. That specification requires that the
 37735 action of the function be atomic.

37736 One of the reasons for introducing this function was to have a means of renaming directories
 37737 while permitting implementations to prohibit the use of *link()* and *unlink()* with directories,
 37738 thus constraining links to directories to those made by *mkdir()*.

37739 The specification that if *old* and *new* refer to the same file is intended to guarantee that:

```
37740 rename("x", "x");
```

37741 does not remove the file.

37742 Renaming dot or dot-dot is prohibited in order to prevent cyclical file system paths.

37743 See also the descriptions of [ENOTEMPTY] and [ENAMETOOLONG] in *rmdir()* and [EBUSY] in
 37744 *unlink()*. For a discussion of [EXDEV], see *link()*.

37745 FUTURE DIRECTIONS

37746 None.

37747 SEE ALSO

37748 *link()*, *rmdir()*, *symlink()*, *unlink()*, the Base Definitions volume of IEEE Std 1003.1-2001,
 37749 `<stdio.h>`

37750 CHANGE HISTORY

37751 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

37752 Issue 5

37753 The [EBUSY] error is added to the “may fail” part of the ERRORS section.

37754 **Issue 6**

37755 Extensions beyond the ISO C standard are marked.

37756 The following new requirements on POSIX implementations derive from alignment with the
37757 Single UNIX Specification:

- 37758 • The [EIO] mandatory error condition is added.
- 37759 • The [ELOOP] mandatory error condition is added.
- 37760 • A second [ENAMETOOLONG] is added as an optional error condition.
- 37761 • The [ETXTBSY] optional error condition is added.

37762 The following changes were made to align with the IEEE P1003.1a draft standard:

- 37763 • Details are added regarding the treatment of symbolic links.
- 37764 • The [ELOOP] optional error condition is added.

37765 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

37766 **NAME**

37767 rewind — reset the file position indicator in a stream

37768 **SYNOPSIS**

37769 #include <stdio.h>

37770 void rewind(FILE *stream);

37771 **DESCRIPTION**

37772 CX The functionality described on this reference page is aligned with the ISO C standard. Any
37773 conflict between the requirements described here and the ISO C standard is unintentional. This
37774 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

37775 The call:

37776 rewind(stream)

37777 shall be equivalent to:

37778 (void) fseek(stream, 0L, SEEK_SET)

37779 except that *rewind()* shall also clear the error indicator.

37780 CX Since *rewind()* does not return a value, an application wishing to detect errors should clear *errno*,
37781 then call *rewind()*, and if *errno* is non-zero, assume an error has occurred.

37782 **RETURN VALUE**

37783 The *rewind()* function shall not return a value.

37784 **ERRORS**

37785 CX Refer to *fseek()* with the exception of [EINVAL] which does not apply.

37786 **EXAMPLES**

37787 None.

37788 **APPLICATION USAGE**

37789 None.

37790 **RATIONALE**

37791 None.

37792 **FUTURE DIRECTIONS**

37793 None.

37794 **SEE ALSO**

37795 *fseek()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

37796 **CHANGE HISTORY**

37797 First released in Issue 1. Derived from Issue 1 of the SVID.

37798 **Issue 6**

37799 Extensions beyond the ISO C standard are marked.

37800 **NAME**

37801 `rewinddir` — reset the position of a directory stream to the beginning of a directory

37802 **SYNOPSIS**

37803 `#include <dirent.h>`

37804 `void rewinddir(DIR *dirp);`

37805 **DESCRIPTION**

37806 The `rewinddir()` function shall reset the position of the directory stream to which *dirp* refers to the beginning of the directory. It shall also cause the directory stream to refer to the current state of the corresponding directory, as a call to `opendir()` would have done. If *dirp* does not refer to a directory stream, the effect is undefined.

37810 After a call to the `fork()` function, either the parent or child (but not both) may continue processing the directory stream using `readdir()`, `rewinddir()`, or `seekdir()`. If both the parent and child processes use these functions, the result is undefined.

37813 **RETURN VALUE**

37814 The `rewinddir()` function shall not return a value.

37815 **ERRORS**

37816 No errors are defined.

37817 **EXAMPLES**

37818 None.

37819 **APPLICATION USAGE**

37820 The `rewinddir()` function should be used in conjunction with `opendir()`, `readdir()`, and `closedir()` to examine the contents of the directory. This method is recommended for portability.

37822 **RATIONALE**

37823 None.

37824 **FUTURE DIRECTIONS**

37825 None.

37826 **SEE ALSO**

37827 `closedir()`, `opendir()`, `readdir()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<dirent.h>`
37828 `<sys/types.h>`

37829 **CHANGE HISTORY**

37830 First released in Issue 2.

37831 **Issue 6**

37832 In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

37833 The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- 37835 • The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

37838 **NAME**37839 rindex — character string operations (**LEGACY**)37840 **SYNOPSIS**

37841 XSI #include <strings.h>

37842 char *rindex(const char *s, int c);

37843

37844 **DESCRIPTION**37845 The *rindex()* function shall be equivalent to *strchr()*.37846 **RETURN VALUE**37847 Refer to *strchr()*.37848 **ERRORS**37849 Refer to *strchr()*.37850 **EXAMPLES**

37851 None.

37852 **APPLICATION USAGE**37853 The *strchr()* function is preferred over this function.37854 For maximum portability, it is recommended to replace the function call to *rindex()* as follows:

37855 #define rindex(a,b) strchr((a),(b))

37856 **RATIONALE**

37857 None.

37858 **FUTURE DIRECTIONS**

37859 This function may be withdrawn in a future version.

37860 **SEE ALSO**37861 *strchr()*, the Base Definitions volume of IEEE Std 1003.1-2001, <strings.h>37862 **CHANGE HISTORY**

37863 First released in Issue 4, Version 2.

37864 **Issue 5**

37865 Moved from X/OPEN UNIX extension to BASE.

37866 **Issue 6**

37867 This function is marked LEGACY.

37868 **NAME**

37869 rint, rintf, rintl — round-to-nearest integral value

37870 **SYNOPSIS**

37871 #include <math.h>

37872 double rint(double x);

37873 float rintf(float x);

37874 long double rintl(long double x);

37875 **DESCRIPTION**

37876 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 37877 conflict between the requirements described here and the ISO C standard is unintentional. This
 37878 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

37879 These functions shall return the integral value (represented as a **double**) nearest *x* in the
 37880 direction of the current rounding mode. The current rounding mode is implementation-defined.

37881 If the current rounding mode rounds toward negative infinity, then *rint()* shall be equivalent to
 37882 *floor()*. If the current rounding mode rounds toward positive infinity, then *rint()* shall be
 37883 equivalent to *ceil()*.

37884 These functions differ from the *nearbyint()*, *nearbyintf()*, and *nearbyintl()* functions only in that
 37885 they may raise the inexact floating-point exception if the result differs in value from the
 37886 argument.

37887 An application wishing to check for error situations should set *errno* to zero and call
 37888 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 37889 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 37890 zero, an error has occurred.

37891 **RETURN VALUE**

37892 Upon successful completion, these functions shall return the integer (represented as a double
 37893 precision number) nearest *x* in the direction of the current rounding mode.

37894 MX If *x* is NaN, a NaN shall be returned.

37895 If *x* is ± 0 or $\pm \text{Inf}$, *x* shall be returned.

37896 XSI If the correct value would cause overflow, a range error shall occur and *rint()*, *rintf()*, and *rintl()*
 37897 shall return the value of the macro $\pm \text{HUGE_VAL}$, $\pm \text{HUGE_VALF}$, and $\pm \text{HUGE_VALL}$ (with the
 37898 same sign as *x*), respectively.

37899 **ERRORS**

37900 These functions shall fail if:

37901 XSI **Range Error** The result would cause an overflow.

37902 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 37903 then *errno* shall be set to [ERANGE]. If the integer expression
 37904 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow
 37905 floating-point exception shall be raised.

37906 **EXAMPLES**

37907 None.

37908 **APPLICATION USAGE**

37909 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
37910 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

37911 **RATIONALE**

37912 None.

37913 **FUTURE DIRECTIONS**

37914 None.

37915 **SEE ALSO**

37916 *abs()*, *ceil()*, *feclearexcept()*, *fetestexcept()*, *floor()*, *isnan()*, *nearbyint()*, the Base Definitions volume
37917 of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
37918 <math.h>

37919 **CHANGE HISTORY**

37920 First released in Issue 4, Version 2.

37921 **Issue 5**

37922 Moved from X/OPEN UNIX extension to BASE.

37923 **Issue 6**

37924 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 37925 • The *rintf()* and *rintl()* functions are added.
- 37926 • The *rint()* function is no longer marked as an extension.
- 37927 • The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
37928 revised to align with the ISO/IEC 9899:1999 standard.
- 37929 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
37930 marked.

37931 **NAME**

37932 rmdir — remove a directory

37933 **SYNOPSIS**

37934 #include <unistd.h>

37935 int rmdir(const char *path);

37936 **DESCRIPTION**

37937 The *rmdir()* function shall remove a directory whose name is given by *path*. The directory shall
 37938 be removed only if it is an empty directory.

37939 If the directory is the root directory or the current working directory of any process, it is
 37940 unspecified whether the function succeeds, or whether it shall fail and set *errno* to [EBUSY].

37941 If *path* names a symbolic link, then *rmdir()* shall fail and set *errno* to [ENOTDIR].

37942 If the *path* argument refers to a path whose final component is either dot or dot-dot, *rmdir()* shall
 37943 fail.

37944 If the directory's link count becomes 0 and no process has the directory open, the space occupied
 37945 by the directory shall be freed and the directory shall no longer be accessible. If one or more
 37946 processes have the directory open when the last link is removed, the dot and dot-dot entries, if
 37947 present, shall be removed before *rmdir()* returns and no new entries may be created in the
 37948 directory, but the directory shall not be removed until all references to the directory are closed.

37949 If the directory is not an empty directory, *rmdir()* shall fail and set *errno* to [EEXIST] or
 37950 [ENOTEMPTY].

37951 Upon successful completion, the *rmdir()* function shall mark for update the *st_ctime* and
 37952 *st_mtime* fields of the parent directory.

37953 **RETURN VALUE**

37954 Upon successful completion, the function *rmdir()* shall return 0. Otherwise, -1 shall be returned,
 37955 and *errno* set to indicate the error. If -1 is returned, the named directory shall not be changed.

37956 **ERRORS**

37957 The *rmdir()* function shall fail if:

37958 [EACCES] Search permission is denied on a component of the path prefix, or write
 37959 permission is denied on the parent directory of the directory to be removed.

37960 [EBUSY] The directory to be removed is currently in use by the system or some process
 37961 and the implementation considers this to be an error.

37962 [EEXIST] or [ENOTEMPTY] The *path* argument names a directory that is not an empty directory, or there
 37963 are hard links to the directory other than dot or a single entry in dot-dot.
 37964

37965 [EINVAL] The *path* argument contains a last component that is dot.

37966 [EIO] A physical I/O error has occurred.

37967 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 37968 argument.

37969 [ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX} or a pathname
 37970 component is longer than {NAME_MAX}.
 37971

37972 [ENOENT] A component of *path* does not name an existing file, or the *path* argument
 37973 names a nonexistent directory or points to an empty string.

37974 [ENOTDIR] A component of *path* is not a directory.

37975 XSI [EPERM] or [EACCES]
 37976 The S_ISVTX flag is set on the parent directory of the directory to be removed
 37977 and the caller is not the owner of the directory to be removed, nor is the caller
 37978 the owner of the parent directory, nor does the caller have the appropriate
 37979 privileges.

37980 [EROFS] The directory entry to be removed resides on a read-only file system.

37981 The *rmdir()* function may fail if:

37982 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 37983 resolution of the *path* argument.

37984 [ENAMETOOLONG]
 37985 As a result of encountering a symbolic link in resolution of the *path* argument,
 37986 the length of the substituted pathname string exceeded {PATH_MAX}.

37987 EXAMPLES

37988 Removing a Directory

37989 The following example shows how to remove a directory named */home/cnd/mod1*.

```
37990 #include <unistd.h>
37991 int status;
37992 ...
37993 status = rmdir("/home/cnd/mod1");
```

37994 APPLICATION USAGE

37995 None.

37996 RATIONALE

37997 The *rmdir()* and *rename()* functions originated in 4.2 BSD, and they used [ENOTEMPTY] for the
 37998 condition when the directory to be removed does not exist or *new* already exists. When the 1984
 37999 /usr/group standard was published, it contained [EEXIST] instead. When these functions were
 38000 adopted into System V, the 1984 /usr/group standard was used as a reference. Therefore, several
 38001 existing applications and implementations support/use both forms, and no agreement could be
 38002 reached on either value. All implementations are required to supply both [EEXIST] and
 38003 [ENOTEMPTY] in *<errno.h>* with distinct values, so that applications can use both values in C-
 38004 language **case** statements.

38005 The meaning of deleting *pathname/dot* is unclear, because the name of the file (directory) in the
 38006 parent directory to be removed is not clear, particularly in the presence of multiple links to a
 38007 directory.

38008 The POSIX.1-1990 standard was silent with regard to the behavior of *rmdir()* when there are
 38009 multiple hard links to the directory being removed. The requirement to set *errno* to [EEXIST] or
 38010 [ENOTEMPTY] clarifies the behavior in this case.

38011 If the process' current working directory is being removed, that should be an allowed error.

38012 Virtually all existing implementations detect [ENOTEMPTY] or the case of dot-dot. The text in
 38013 Section 2.3 (on page 21) about returning any one of the possible errors permits that behavior to
 38014 continue. The [ELOOP] error may be returned if more than {SYMLOOP_MAX} symbolic links
 38015 are encountered during resolution of the *path* argument.

38016 **FUTURE DIRECTIONS**

38017 None.

38018 **SEE ALSO**

38019 Section 2.3 (on page 21), *mkdir()*, *remove()*, *unlink()*, the Base Definitions volume of
38020 IEEE Std 1003.1-2001, <unistd.h>

38021 **CHANGE HISTORY**

38022 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

38023 **Issue 6**

38024 The following new requirements on POSIX implementations derive from alignment with the
38025 Single UNIX Specification:

- 38026 • The DESCRIPTION is updated to indicate the results of naming a symbolic link in *path*.
- 38027 • The [EIO] mandatory error condition is added.
- 38028 • The [ELOOP] mandatory error condition is added.
- 38029 • A second [ENAMETOOLONG] is added as an optional error condition.

38030 The following changes were made to align with the IEEE P1003.1a draft standard:

- 38031 • The [ELOOP] optional error condition is added.

38032 **NAME**

38033 round, roundf, roundl — round to the nearest integer value in a floating-point format

38034 **SYNOPSIS**

38035 #include <math.h>

38036 double round(double x);

38037 float roundf(float x);

38038 long double roundl(long double x);

38039 **DESCRIPTION**

38040 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 38041 conflict between the requirements described here and the ISO C standard is unintentional. This
 38042 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

38043 These functions shall round their argument to the nearest integer value in floating-point format,
 38044 rounding halfway cases away from zero, regardless of the current rounding direction.

38045 An application wishing to check for error situations should set *errno* to zero and call
 38046 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 38047 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 38048 zero, an error has occurred.

38049 **RETURN VALUE**

38050 Upon successful completion, these functions shall return the rounded integer value.

38051 MX If *x* is NaN, a NaN shall be returned.38052 If *x* is ± 0 or $\pm \text{Inf}$, *x* shall be returned.

38053 XSI If the correct value would cause overflow, a range error shall occur and *round()*, *roundf()*, and
 38054 *roundl()* shall return the value of the macro $\pm \text{HUGE_VAL}$, $\pm \text{HUGE_VALF}$, and $\pm \text{HUGE_VALL}$
 38055 (with the same sign as *x*), respectively.

38056 **ERRORS**

38057 These functions may fail if:

38058 XSI **Range Error** The result overflows.

38059 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 38060 then *errno* shall be set to [ERANGE]. If the integer expression
 38061 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow
 38062 floating-point exception shall be raised.

38063 **EXAMPLES**

38064 None.

38065 **APPLICATION USAGE**

38066 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
 38067 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

38068 **RATIONALE**

38069 None.

38070 **FUTURE DIRECTIONS**

38071 None.

38072 **SEE ALSO**

38073 *feclearexcept()*, *fetestexcept()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18,
38074 Treatment of Error Conditions for Mathematical Functions, <**math.h**>

38075 **CHANGE HISTORY**

38076 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

38077 **NAME**38078 `scalb` — load exponent of a radix-independent floating-point number38079 **SYNOPSIS**38080 OB XSI `#include <math.h>`38081 `double scalb(double x, double n);`

38082

38083 **DESCRIPTION**

38084 The `scalb()` function shall compute $x \cdot r^n$, where r is the radix of the machine's floating-point arithmetic. When r is 2, `scalb()` shall be equivalent to `ldexp()`. The value of r is `FLT_RADIX` which is defined in `<float.h>`.

38087 An application wishing to check for error situations should set `errno` to zero and call `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `errno` is non-zero or `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-zero, an error has occurred.

38091 **RETURN VALUE**38092 Upon successful completion, the `scalb()` function shall return $x \cdot r^n$.38093 If x or n is NaN, a NaN shall be returned.38094 If n is zero, x shall be returned.38095 If x is $\pm\text{Inf}$ and n is not $-\text{Inf}$, x shall be returned.38096 If x is ± 0 and n is not $+\text{Inf}$, x shall be returned.

38097 If x is ± 0 and n is $+\text{Inf}$, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

38099 If x is $\pm\text{Inf}$ and n is $-\text{Inf}$, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

38101 If the result would cause an overflow, a range error shall occur and $\pm\text{HUGE_VAL}$ (according to the sign of x) shall be returned.

38103 If the correct value would cause underflow, and is representable, a range error may occur and the correct value shall be returned.

38105 If the correct value would cause underflow, and is not representable, a range error may occur, and 0.0 shall be returned.

38107 **ERRORS**38108 The `scalb()` function shall fail if:38109 Domain Error If x is zero and n is $+\text{Inf}$, or x is Inf and n is $-\text{Inf}$.

38110 If the integer expression `(math_errhandling & MATH_ERRNO)` is non-zero, then `errno` shall be set to `[EDOM]`. If the integer expression `(math_errhandling & MATH_ERREXCEPT)` is non-zero, then the invalid floating-point exception shall be raised.

38114 Range Error The result would overflow.

38115 If the integer expression `(math_errhandling & MATH_ERRNO)` is non-zero, then `errno` shall be set to `[ERANGE]`. If the integer expression `(math_errhandling & MATH_ERREXCEPT)` is non-zero, then the overflow floating-point exception shall be raised.

38119 The *scalb()* function may fail if:

38120 Range Error The result underflows.

38121 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 38122 then *errno* shall be set to [ERANGE]. If the integer expression
 38123 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
 38124 floating-point exception shall be raised.

38125 EXAMPLES

38126 None.

38127 APPLICATION USAGE

38128 Applications should use either *scalbln()*, *scalblnf()*, or *scalblnl()* in preference to this function.

38129 IEEE Std 1003.1-2001 only defines the behavior for the *scalb()* function when the *n* argument is
 38130 an integer, a NaN, or Inf. The behavior of other values for the *n* argument is unspecified.

38131 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
 38132 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

38133 RATIONALE

38134 None.

38135 FUTURE DIRECTIONS

38136 None.

38137 SEE ALSO

38138 *feclearexcept()*, *fetestexcept()*, *ilogb()*, *ldexp()*, *logb()*, *scalbln()*, the Base Definitions volume of
 38139 IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
 38140 <float.h>, <math.h>

38141 CHANGE HISTORY

38142 First released in Issue 4, Version 2.

38143 Issue 5

38144 Moved from X/OPEN UNIX extension to BASE.

38145 The DESCRIPTION is updated to indicate how an application should check for an error. This
 38146 text was previously published in the APPLICATION USAGE section.

38147 Issue 6

38148 This function is marked obsolescent.

38149 Although this function is not part of the ISO/IEC 9899:1999 standard, the RETURN VALUE and
 38150 ERRORS sections are updated to align with the error handling in the ISO/IEC 9899:1999
 38151 standard.

38152 **NAME**38153 `scalbln, scalblnf, scalblnl, scalbn, scalbnf, scalbnl`, — compute exponent using FLT_RADIX38154 **SYNOPSIS**38155 `#include <math.h>`

```

38156        double scalbln(double x, long n);
38157        float scalblnf(float x, long n);
38158        long double scalblnl(long double x, long n);
38159        double scalbn(double x, int n);
38160        float scalbnf(float x, int n);
38161        long double scalbnl(long double x, int n);

```

38162 **DESCRIPTION**

38163 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 38164 conflict between the requirements described here and the ISO C standard is unintentional. This
 38165 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

38166 These functions shall compute $x * \text{FLT_RADIX}^n$ efficiently, not normally by computing
 38167 FLT_RADIX^n explicitly.

38168 An application wishing to check for error situations should set *errno* to zero and call
 38169 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 38170 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 38171 zero, an error has occurred.

38172 **RETURN VALUE**38173 Upon successful completion, these functions shall return $x * \text{FLT_RADIX}^n$.

38174 If the result would cause overflow, a range error shall occur and these functions shall return
 38175 $\pm\text{HUGE_VAL}$, $\pm\text{HUGE_VALF}$, and $\pm\text{HUGE_VALL}$ (according to the sign of *x*) as appropriate for
 38176 the return type of the function.

38177 If the correct value would cause underflow, and is not representable, a range error may occur,
 38178 **MX** and either 0.0 (if supported), or an implementation-defined value shall be returned.

38179 **MX** If *x* is NaN, a NaN shall be returned.38180 If *x* is ± 0 or $\pm\text{Inf}$, *x* shall be returned.38181 If *n* is 0, *x* shall be returned.

38182 If the correct value would cause underflow, and is representable, a range error may occur and
 38183 the correct value shall be returned.

38184 **ERRORS**

38185 These functions shall fail if:

38186 Range Error The result overflows.

38187 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 38188 then *errno* shall be set to [ERANGE]. If the integer expression
 38189 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
 38190 floating-point exception shall be raised.

38191 These functions may fail if:

38192 Range Error The result underflows.

38193 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 38194 then *errno* shall be set to [ERANGE]. If the integer expression

38195 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
38196 floating-point exception shall be raised.

38197 EXAMPLES

38198 None.

38199 APPLICATION USAGE

38200 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
38201 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

38202 RATIONALE

38203 These functions are named so as to avoid conflicting with the historical definition of the *scalb()*
38204 function from the Single UNIX Specification. The difference is that the *scalb()* function has a
38205 second argument of **double** instead of **int**. The *scalb()* function is not part of the ISO C standard.
38206 The three functions whose second type is **long** are provided because the factor required to scale
38207 from the smallest positive floating-point value to the largest finite one, on many
38208 implementations, is too large to represent in the minimum-width **int** format.

38209 FUTURE DIRECTIONS

38210 None.

38211 SEE ALSO

38212 *feclearexcept()*, *fetestexcept()*, *scalb()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section
38213 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

38214 CHANGE HISTORY

38215 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

38216 **NAME**

38217 scanf — convert formatted input

38218 **SYNOPSIS**

38219 #include <stdio.h>

38220 int scanf(const char *restrict *format*, ...);38221 **DESCRIPTION**38222 Refer to *fscanf()*.

38223 NAME

38224 sched_get_priority_max, sched_get_priority_min — get priority limits (**REALTIME**)

38225 SYNOPSIS

38226 PS `#include <sched.h>`

38227 `int sched_get_priority_max(int policy);`

38228 `int sched_get_priority_min(int policy);`

38229

38230 DESCRIPTION

38231 The *sched_get_priority_max()* and *sched_get_priority_min()* functions shall return the appropriate
38232 maximum or minimum, respectively, for the scheduling policy specified by *policy*.

38233 The value of *policy* shall be one of the scheduling policy values defined in **<sched.h>**.

38234 RETURN VALUE

38235 If successful, the *sched_get_priority_max()* and *sched_get_priority_min()* functions shall return the
38236 appropriate maximum or minimum values, respectively. If unsuccessful, they shall return a
38237 value of `-1` and set *errno* to indicate the error.

38238 ERRORS

38239 The *sched_get_priority_max()* and *sched_get_priority_min()* functions shall fail if:

38240 [EINVAL] The value of the *policy* parameter does not represent a defined scheduling
38241 policy.

38242 EXAMPLES

38243 None.

38244 APPLICATION USAGE

38245 None.

38246 RATIONALE

38247 None.

38248 FUTURE DIRECTIONS

38249 None.

38250 SEE ALSO

38251 *sched_getparam()*, *sched_setparam()*, *sched_getscheduler()*, *sched_rr_get_interval()*,
38252 *sched_setscheduler()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<sched.h>**

38253 CHANGE HISTORY

38254 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38255 Issue 6

38256 These functions are marked as part of the Process Scheduling option.

38257 The [ENOSYS] error condition has been removed as stubs need not be provided if an
38258 implementation does not support the Process Scheduling option.

38259 The [ESRCH] error condition has been removed since these functions do not take a *pid*
38260 argument.

38261 **NAME**38262 sched_getparam — get scheduling parameters (**REALTIME**)38263 **SYNOPSIS**

38264 PS #include <sched.h>

38265 int sched_getparam(pid_t pid, struct sched_param *param);

38266

38267 **DESCRIPTION**38268 The *sched_getparam()* function shall return the scheduling parameters of a process specified by
38269 *pid* in the **sched_param** structure pointed to by *param*.38270 If a process specified by *pid* exists, and if the calling process has permission, the scheduling
38271 parameters for the process whose process ID is equal to *pid* shall be returned.38272 If *pid* is zero, the scheduling parameters for the calling process shall be returned. The behavior of
38273 the *sched_getparam()* function is unspecified if the value of *pid* is negative.38274 **RETURN VALUE**38275 Upon successful completion, the *sched_getparam()* function shall return zero. If the call to
38276 *sched_getparam()* is unsuccessful, the function shall return a value of -1 and set *errno* to indicate
38277 the error.38278 **ERRORS**38279 The *sched_getparam()* function shall fail if:38280 [EPERM] The requesting process does not have permission to obtain the scheduling
38281 parameters of the specified process.38282 [ESRCH] No process can be found corresponding to that specified by *pid*.38283 **EXAMPLES**

38284 None.

38285 **APPLICATION USAGE**

38286 None.

38287 **RATIONALE**

38288 None.

38289 **FUTURE DIRECTIONS**

38290 None.

38291 **SEE ALSO**38292 *sched_getscheduler()*, *sched_setparam()*, *sched_setscheduler()*, the Base Definitions volume of
38293 IEEE Std 1003.1-2001, <**sched.h**>38294 **CHANGE HISTORY**

38295 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38296 **Issue 6**38297 The *sched_getparam()* function is marked as part of the Process Scheduling option.38298 The [ENOSYS] error condition has been removed as stubs need not be provided if an
38299 implementation does not support the Process Scheduling option.

38300 NAME

38301 sched_getscheduler — get scheduling policy (**REALTIME**)

38302 SYNOPSIS

38303 PS `#include <sched.h>`

38304 `int sched_getscheduler(pid_t pid);`

38305

38306 DESCRIPTION

38307 The *sched_getscheduler()* function shall return the scheduling policy of the process specified by
38308 *pid*. If the value of *pid* is negative, the behavior of the *sched_getscheduler()* function is
38309 unspecified.

38310 The values that can be returned by *sched_getscheduler()* are defined in the **<sched.h>** header.

38311 If a process specified by *pid* exists, and if the calling process has permission, the scheduling
38312 policy shall be returned for the process whose process ID is equal to *pid*.

38313 If *pid* is zero, the scheduling policy shall be returned for the calling process.

38314 RETURN VALUE

38315 Upon successful completion, the *sched_getscheduler()* function shall return the scheduling policy
38316 of the specified process. If unsuccessful, the function shall return `-1` and set *errno* to indicate the
38317 error.

38318 ERRORS

38319 The *sched_getscheduler()* function shall fail if:

38320 [EPERM] The requesting process does not have permission to determine the scheduling
38321 policy of the specified process.

38322 [ESRCH] No process can be found corresponding to that specified by *pid*.

38323 EXAMPLES

38324 None.

38325 APPLICATION USAGE

38326 None.

38327 RATIONALE

38328 None.

38329 FUTURE DIRECTIONS

38330 None.

38331 SEE ALSO

38332 *sched_getparam()*, *sched_setparam()*, *sched_setscheduler()*, the Base Definitions volume of
38333 IEEE Std 1003.1-2001, **<sched.h>**

38334 CHANGE HISTORY

38335 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38336 Issue 6

38337 The *sched_getscheduler()* function is marked as part of the Process Scheduling option.

38338 The [ENOSYS] error condition has been removed as stubs need not be provided if an
38339 implementation does not support the Process Scheduling option.

38340 **NAME**38341 sched_rr_get_interval — get execution time limits (**REALTIME**)38342 **SYNOPSIS**38343 PS `#include <sched.h>`38344 `int sched_rr_get_interval(pid_t pid, struct timespec *interval);`

38345

38346 **DESCRIPTION**

38347 The *sched_rr_get_interval()* function shall update the **timespec** structure referenced by the
38348 *interval* argument to contain the current execution time limit (that is, time quantum) for the
38349 process specified by *pid*. If *pid* is zero, the current execution time limit for the calling process
38350 shall be returned.

38351 **RETURN VALUE**

38352 If successful, the *sched_rr_get_interval()* function shall return zero. Otherwise, it shall return a
38353 value of -1 and set *errno* to indicate the error.

38354 **ERRORS**38355 The *sched_rr_get_interval()* function shall fail if:

38356 [ESRCH] No process can be found corresponding to that specified by *pid*.

38357 **EXAMPLES**

38358 None.

38359 **APPLICATION USAGE**

38360 None.

38361 **RATIONALE**

38362 None.

38363 **FUTURE DIRECTIONS**

38364 None.

38365 **SEE ALSO**

38366 *sched_getparam()*, *sched_get_priority_max()*, *sched_getscheduler()*, *sched_setparam()*,
38367 *sched_setscheduler()*, the Base Definitions volume of IEEE Std 1003.1-2001, `<sched.h>`

38368 **CHANGE HISTORY**

38369 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38370 **Issue 6**38371 The *sched_rr_get_interval()* function is marked as part of the Process Scheduling option.

38372 The [ENOSYS] error condition has been removed as stubs need not be provided if an
38373 implementation does not support the Process Scheduling option.

38374 NAME

38375 sched_setparam — set scheduling parameters (**REALTIME**)

38376 SYNOPSIS

38377 PS `#include <sched.h>`

38378 `int sched_setparam(pid_t pid, const struct sched_param *param);`

38379

38380 DESCRIPTION

38381 The *sched_setparam()* function shall set the scheduling parameters of the process specified by *pid*
 38382 to the values specified by the **sched_param** structure pointed to by *param*. The value of the
 38383 *sched_priority* member in the **sched_param** structure shall be any integer within the inclusive
 38384 priority range for the current scheduling policy of the process specified by *pid*. Higher
 38385 numerical values for the priority represent higher priorities. If the value of *pid* is negative, the
 38386 behavior of the *sched_setparam()* function is unspecified.

38387 If a process specified by *pid* exists, and if the calling process has permission, the scheduling
 38388 parameters shall be set for the process whose process ID is equal to *pid*.

38389 If *pid* is zero, the scheduling parameters shall be set for the calling process.

38390 The conditions under which one process has permission to change the scheduling parameters of
 38391 another process are implementation-defined.

38392 Implementations may require the requesting process to have the appropriate privilege to set its
 38393 own scheduling parameters or those of another process.

38394 The target process, whether it is running or not running, shall be moved to the tail of the thread
 38395 list for its priority.

38396 If the priority of the process specified by the *pid* argument is set higher than that of the lowest
 38397 priority running process and if the specified process is ready to run, the process specified by the
 38398 *pid* argument shall preempt a lowest priority running process. Similarly, if the process calling
 38399 *sched_setparam()* sets its own priority lower than that of one or more other non-empty process
 38400 lists, then the process that is the head of the highest priority list shall also preempt the calling
 38401 process. Thus, in either case, the originating process might not receive notification of the
 38402 completion of the requested priority change until the higher priority process has executed.

38403 ss If the scheduling policy of the target process is SCHED_SPORADIC, the value specified by the
 38404 *sched_ss_low_priority* member of the *param* argument shall be any integer within the inclusive
 38405 priority range for the sporadic server policy. The *sched_ss_repl_period* and *sched_ss_init_budget*
 38406 members of the *param* argument shall represent the time parameters to be used by the sporadic
 38407 server scheduling policy for the target process. The *sched_ss_max_repl* member of the *param*
 38408 argument shall represent the maximum number of replenishments that are allowed to be
 38409 pending simultaneously for the process scheduled under this scheduling policy.

38410 The specified *sched_ss_repl_period* shall be greater than or equal to the specified
 38411 *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail.

38412 The value of *sched_ss_max_repl* shall be within the inclusive range [1,{SS_REPL_MAX}] for the
 38413 function to succeed; if not, the function shall fail.

38414 If the scheduling policy of the target process is either SCHED_FIFO or SCHED_RR, the
 38415 *sched_ss_low_priority*, *sched_ss_repl_period*, and *sched_ss_init_budget* members of the *param*
 38416 argument shall have no effect on the scheduling behavior. If the scheduling policy of this process
 38417 is not SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC, the effects of these members are
 38418 implementation-defined; this case includes the SCHED_OTHER policy.

38419 If the current scheduling policy for the process specified by *pid* is not SCHED_FIFO,
 38420 ss SCHED_RR, or SCHED_SPORADIC, the result is implementation-defined; this case includes the
 38421 SCHED_OTHER policy.

38422 The effect of this function on individual threads is dependent on the scheduling contention
 38423 scope of the threads:

- 38424 • For threads with system scheduling contention scope, these functions shall have no effect on
 38425 their scheduling.
- 38426 • For threads with process scheduling contention scope, the threads' scheduling parameters
 38427 shall not be affected. However, the scheduling of these threads with respect to threads in
 38428 other processes may be dependent on the scheduling parameters of their process, which are
 38429 governed using these functions.

38430 If an implementation supports a two-level scheduling model in which library threads are
 38431 multiplexed on top of several kernel-scheduled entities, then the underlying kernel-scheduled
 38432 entities for the system contention scope threads shall not be affected by these functions.

38433 The underlying kernel-scheduled entities for the process contention scope threads shall have
 38434 their scheduling parameters changed to the value specified in *param*. Kernel-scheduled entities
 38435 for use by process contention scope threads that are created after this call completes shall inherit
 38436 their scheduling policy and associated scheduling parameters from the process.

38437 This function is not atomic with respect to other threads in the process. Threads may continue to
 38438 execute while this function call is in the process of changing the scheduling policy for the
 38439 underlying kernel-scheduled entities used by the process contention scope threads.

38440 RETURN VALUE

38441 If successful, the *sched_setparam()* function shall return zero.

38442 If the call to *sched_setparam()* is unsuccessful, the priority shall remain unchanged, and the
 38443 function shall return a value of -1 and set *errno* to indicate the error.

38444 ERRORS

38445 The *sched_setparam()* function shall fail if:

- | | | |
|-------|----------|---|
| 38446 | [EINVAL] | One or more of the requested scheduling parameters is outside the range |
| 38447 | | defined for the scheduling policy of the specified <i>pid</i> . |
| 38448 | [EPERM] | The requesting process does not have permission to set the scheduling |
| 38449 | | parameters for the specified process, or does not have the appropriate |
| 38450 | | privilege to invoke <i>sched_setparam()</i> . |
| 38451 | [ESRCH] | No process can be found corresponding to that specified by <i>pid</i> . |

38452 EXAMPLES

38453 None.

38454 APPLICATION USAGE

38455 None.

38456 RATIONALE

38457 None.

38458 FUTURE DIRECTIONS

38459 None.

38460 SEE ALSO

38461 *sched_getparam()*, *sched_getscheduler()*, *sched_setscheduler()*, the Base Definitions volume of
 38462 IEEE Std 1003.1-2001, <**sched.h**>

38463 CHANGE HISTORY

38464 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38465 Issue 6

38466 The *sched_setparam()* function is marked as part of the Process Scheduling option.

38467 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 38468 implementation does not support the Process Scheduling option.

38469 The following new requirements on POSIX implementations derive from alignment with the
 38470 Single UNIX Specification:

38471 • In the DESCRIPTION, the effect of this function on a thread's scheduling parameters is
 38472 added.

38473 • Sections describing two-level scheduling and atomicity of the function are added.

38474 The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

38475 IEEE PASC Interpretation 1003.1 #100 is applied.

38476 NAME

38477 sched_setscheduler — set scheduling policy and parameters (**REALTIME**)

38478 SYNOPSIS

38479 PS `#include <sched.h>`

```
38480 int sched_setscheduler(pid_t pid, int policy,
38481     const struct sched_param *param);
38482
```

38483 DESCRIPTION

38484 The *sched_setscheduler()* function shall set the scheduling policy and scheduling parameters of
 38485 the process specified by *pid* to *policy* and the parameters specified in the **sched_param** structure
 38486 pointed to by *param*, respectively. The value of the *sched_priority* member in the **sched_param**
 38487 structure shall be any integer within the inclusive priority range for the scheduling policy
 38488 specified by *policy*. If the value of *pid* is negative, the behavior of the *sched_setscheduler()*
 38489 function is unspecified.

38490 The possible values for the *policy* parameter are defined in the **<sched.h>** header.

38491 If a process specified by *pid* exists, and if the calling process has permission, the scheduling
 38492 policy and scheduling parameters shall be set for the process whose process ID is equal to *pid*.

38493 If *pid* is zero, the scheduling policy and scheduling parameters shall be set for the calling
 38494 process.

38495 The conditions under which one process has the appropriate privilege to change the scheduling
 38496 parameters of another process are implementation-defined.

38497 Implementations may require that the requesting process have permission to set its own
 38498 scheduling parameters or those of another process. Additionally, implementation-defined
 38499 restrictions may apply as to the appropriate privileges required to set a process' own scheduling
 38500 policy, or another process' scheduling policy, to a particular value.

38501 The *sched_setscheduler()* function shall be considered successful if it succeeds in setting the
 38502 scheduling policy and scheduling parameters of the process specified by *pid* to the values
 38503 specified by *policy* and the structure pointed to by *param*, respectively.

38504 SS If the scheduling policy specified by *policy* is SCHED_SPORADIC, the value specified by the
 38505 *sched_ss_low_priority* member of the *param* argument shall be any integer within the inclusive
 38506 priority range for the sporadic server policy. The *sched_ss_repl_period* and *sched_ss_init_budget*
 38507 members of the *param* argument shall represent the time parameters used by the sporadic server
 38508 scheduling policy for the target process. The *sched_ss_max_repl* member of the *param* argument
 38509 shall represent the maximum number of replenishments that are allowed to be pending
 38510 simultaneously for the process scheduled under this scheduling policy.

38511 The specified *sched_ss_repl_period* shall be greater than or equal to the specified
 38512 *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail.

38513 The value of *sched_ss_max_repl* shall be within the inclusive range [1,{SS_REPL_MAX}] for the
 38514 function to succeed; if not, the function shall fail.

38515 If the scheduling policy specified by *policy* is either SCHED_FIFO or SCHED_RR, the
 38516 *sched_ss_low_priority*, *sched_ss_repl_period*, and *sched_ss_init_budget* members of the *param*
 38517 argument shall have no effect on the scheduling behavior.

38518 The effect of this function on individual threads is dependent on the scheduling contention
 38519 scope of the threads:

38520 • For threads with system scheduling contention scope, these functions shall have no effect on
38521 their scheduling.

38522 • For threads with process scheduling contention scope, the threads' scheduling policy and
38523 associated parameters shall not be affected. However, the scheduling of these threads with
38524 respect to threads in other processes may be dependent on the scheduling parameters of their
38525 process, which are governed using these functions.

38526 If an implementation supports a two-level scheduling model in which library threads are
38527 multiplexed on top of several kernel-scheduled entities, then the underlying kernel-scheduled
38528 entities for the system contention scope threads shall not be affected by these functions.

38529 The underlying kernel-scheduled entities for the process contention scope threads shall have
38530 their scheduling policy and associated scheduling parameters changed to the values specified in
38531 *policy* and *param*, respectively. Kernel-scheduled entities for use by process contention scope
38532 threads that are created after this call completes shall inherit their scheduling policy and
38533 associated scheduling parameters from the process.

38534 This function is not atomic with respect to other threads in the process. Threads may continue to
38535 execute while this function call is in the process of changing the scheduling policy and
38536 associated scheduling parameters for the underlying kernel-scheduled entities used by the
38537 process contention scope threads.

38538 RETURN VALUE

38539 Upon successful completion, the function shall return the former scheduling policy of the
38540 specified process. If the *sched_setscheduler()* function fails to complete successfully, the policy
38541 and scheduling parameters shall remain unchanged, and the function shall return a value of *-1*
38542 and set *errno* to indicate the error.

38543 ERRORS

38544 The *sched_setscheduler()* function shall fail if:

38545 [EINVAL] The value of the *policy* parameter is invalid, or one or more of the parameters
38546 contained in *param* is outside the valid range for the specified scheduling
38547 policy.

38548 [EPERM] The requesting process does not have permission to set either or both of the
38549 scheduling parameters or the scheduling policy of the specified process.

38550 [ESRCH] No process can be found corresponding to that specified by *pid*.

38551 EXAMPLES

38552 None.

38553 APPLICATION USAGE

38554 None.

38555 RATIONALE

38556 None.

38557 FUTURE DIRECTIONS

38558 None.

38559 SEE ALSO

38560 *sched_getparam()*, *sched_getscheduler()*, *sched_setparam()*, the Base Definitions volume of
38561 IEEE Std 1003.1-2001, <**sched.h**>

38562 **CHANGE HISTORY**

38563 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38564 **Issue 6**

38565 The *sched_setscheduler()* function is marked as part of the Process Scheduling option.

38566 The [ENOSYS] error condition has been removed as stubs need not be provided if an
38567 implementation does not support the Process Scheduling option.

38568 The following new requirements on POSIX implementations derive from alignment with the
38569 Single UNIX Specification:

38570 • In the DESCRIPTION, the effect of this function on a thread's scheduling parameters is
38571 added.

38572 • Sections describing two-level scheduling and atomicity of the function are added.

38573 The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

38574 NAME

38575 sched_yield — yield the processor

38576 SYNOPSIS

38577 PS|THR `#include <sched.h>`

38578 `int sched_yield(void);`

38579

38580 DESCRIPTION

38581 The *sched_yield()* function shall force the running thread to relinquish the processor until it again
38582 becomes the head of its thread list. It takes no arguments.

38583 RETURN VALUE

38584 The *sched_yield()* function shall return 0 if it completes successfully; otherwise, it shall return a
38585 value of -1 and set *errno* to indicate the error.

38586 ERRORS

38587 No errors are defined.

38588 EXAMPLES

38589 None.

38590 APPLICATION USAGE

38591 None.

38592 RATIONALE

38593 None.

38594 FUTURE DIRECTIONS

38595 None.

38596 SEE ALSO

38597 The Base Definitions volume of IEEE Std 1003.1-2001, <**sched.h**>

38598 CHANGE HISTORY

38599 First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the
38600 POSIX Threads Extension.

38601 Issue 6

38602 The *sched_yield()* function is now marked as part of the Process Scheduling and Threads options.

38603 **NAME**

38604 seed48 — seed a uniformly distributed pseudo-random non-negative long integer generator

38605 **SYNOPSIS**

38606 xSI #include <stdlib.h>

38607 unsigned short *seed48(unsigned short *seed16v*[3]);

38608

38609 **DESCRIPTION**

38610 Refer to *drand48()*.

38611 **NAME**

38612 seekdir — set the position of a directory stream

38613 **SYNOPSIS**38614 XSI `#include <dirent.h>`38615 `void seekdir(DIR *dirp, long loc);`

38616

38617 **DESCRIPTION**

38618 The *seekdir()* function shall set the position of the next *readdir()* operation on the directory
38619 stream specified by *dirp* to the position specified by *loc*. The value of *loc* should have been
38620 returned from an earlier call to *telldir()*. The new position reverts to the one associated with the
38621 directory stream when *telldir()* was performed.

38622 If the value of *loc* was not obtained from an earlier call to *telldir()*, or if a call to *rewinddir()*
38623 occurred between the call to *telldir()* and the call to *seekdir()*, the results of subsequent calls to
38624 *readdir()* are unspecified.

38625 **RETURN VALUE**38626 The *seekdir()* function shall not return a value.38627 **ERRORS**

38628 No errors are defined.

38629 **EXAMPLES**

38630 None.

38631 **APPLICATION USAGE**

38632 None.

38633 **RATIONALE**

38634 The original standard developers perceived that there were restrictions on the use of the
38635 *seekdir()* and *telldir()* functions related to implementation details, and for that reason these
38636 functions need not be supported on all POSIX-conforming systems. They are required on
38637 implementations supporting the XSI extension.

38638 One of the perceived problems of implementation is that returning to a given point in a directory
38639 is quite difficult to describe formally, in spite of its intuitive appeal, when systems that use B-
38640 trees, hashing functions, or other similar mechanisms to order their directories are considered.
38641 The definition of *seekdir()* and *telldir()* does not specify whether, when using these interfaces, a
38642 given directory entry will be seen at all, or more than once.

38643 On systems not supporting these functions, their capability can sometimes be accomplished by
38644 saving a filename found by *readdir()* and later using *rewinddir()* and a loop on *readdir()* to
38645 relocate the position from which the filename was saved.

38646 **FUTURE DIRECTIONS**

38647 None.

38648 **SEE ALSO**

38649 *opendir()*, *readdir()*, *telldir()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<dirent.h>**,
38650 **<stdio.h>**, **<sys/types.h>**

38651 **CHANGE HISTORY**

38652 First released in Issue 2.

38653 **Issue 6**

38654 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

38655 NAME

38656 select — synchronous I/O multiplexing

38657 SYNOPSIS

38658 #include <sys/time.h>

38659 int select(int *nfds*, fd_set *restrict *readfds*,
38660 fd_set *restrict *writefds*, fd_set *restrict *errorfds*,
38661 struct timeval *restrict *timeout*);
38662

38663 DESCRIPTION

38664 Refer to *pselect()*.

38665 **NAME**38666 `sem_close` — close a named semaphore (**REALTIME**)38667 **SYNOPSIS**38668 SEM `#include <semaphore.h>`38669 `int sem_close(sem_t *sem);`

38670

38671 **DESCRIPTION**

38672 The `sem_close()` function shall indicate that the calling process is finished using the named
 38673 semaphore indicated by `sem`. The effects of calling `sem_close()` for an unnamed semaphore (one
 38674 created by `sem_init()`) are undefined. The `sem_close()` function shall deallocate (that is, make
 38675 available for reuse by a subsequent `sem_open()` by this process) any system resources allocated
 38676 by the system for use by this process for this semaphore. The effect of subsequent use of the
 38677 semaphore indicated by `sem` by this process is undefined. If the semaphore has not been
 38678 removed with a successful call to `sem_unlink()`, then `sem_close()` has no effect on the state of the
 38679 semaphore. If the `sem_unlink()` function has been successfully invoked for `name` after the most
 38680 recent call to `sem_open()` with `O_CREAT` for this semaphore, then when all processes that have
 38681 opened the semaphore close it, the semaphore is no longer accessible.

38682 **RETURN VALUE**

38683 Upon successful completion, a value of zero shall be returned. Otherwise, a value of `-1` shall be
 38684 returned and `errno` set to indicate the error.

38685 **ERRORS**38686 The `sem_close()` function shall fail if:38687 [EINVAL] The `sem` argument is not a valid semaphore descriptor.38688 **EXAMPLES**

38689 None.

38690 **APPLICATION USAGE**

38691 The `sem_close()` function is part of the Semaphores option and need not be available on all
 38692 implementations.

38693 **RATIONALE**

38694 None.

38695 **FUTURE DIRECTIONS**

38696 None.

38697 **SEE ALSO**

38698 `semctl()`, `semget()`, `semop()`, `sem_init()`, `sem_open()`, `sem_unlink()`, the Base Definitions volume of
 38699 IEEE Std 1003.1-2001, `<semaphore.h>`

38700 **CHANGE HISTORY**

38701 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38702 **Issue 6**38703 The `sem_close()` function is marked as part of the Semaphores option.

38704 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 38705 implementation does not support the Semaphores option.

38706 **NAME**38707 sem_destroy — destroy an unnamed semaphore (**REALTIME**)38708 **SYNOPSIS**38709 SEM `#include <semaphore.h>`38710 `int sem_destroy(sem_t *sem);`

38711

38712 **DESCRIPTION**

38713 The `sem_destroy()` function shall destroy the unnamed semaphore indicated by `sem`. Only a
38714 semaphore that was created using `sem_init()` may be destroyed using `sem_destroy()`; the effect of
38715 calling `sem_destroy()` with a named semaphore is undefined. The effect of subsequent use of the
38716 semaphore `sem` is undefined until `sem` is reinitialized by another call to `sem_init()`.

38717 It is safe to destroy an initialized semaphore upon which no threads are currently blocked. The
38718 effect of destroying a semaphore upon which other threads are currently blocked is undefined.

38719 **RETURN VALUE**

38720 Upon successful completion, a value of zero shall be returned. Otherwise, a value of `-1` shall be
38721 returned and `errno` set to indicate the error.

38722 **ERRORS**38723 The `sem_destroy()` function shall fail if:38724 [EINVAL] The `sem` argument is not a valid semaphore.38725 The `sem_destroy()` function may fail if:

38726 [EBUSY] There are currently processes blocked on the semaphore.

38727 **EXAMPLES**

38728 None.

38729 **APPLICATION USAGE**

38730 The `sem_destroy()` function is part of the Semaphores option and need not be available on all
38731 implementations.

38732 **RATIONALE**

38733 None.

38734 **FUTURE DIRECTIONS**

38735 None.

38736 **SEE ALSO**

38737 `semctl()`, `semget()`, `semop()`, `sem_init()`, `sem_open()`, the Base Definitions volume of
38738 IEEE Std 1003.1-2001, `<semaphore.h>`

38739 **CHANGE HISTORY**

38740 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38741 **Issue 6**38742 The `sem_destroy()` function is marked as part of the Semaphores option.

38743 The [ENOSYS] error condition has been removed as stubs need not be provided if an
38744 implementation does not support the Semaphores option.

38745 **NAME**38746 `sem_getvalue` — get the value of a semaphore (**REALTIME**)38747 **SYNOPSIS**38748 SEM `#include <semaphore.h>`38749 `int sem_getvalue(sem_t *restrict sem, int *restrict sval);`

38750

38751 **DESCRIPTION**

38752 The `sem_getvalue()` function shall update the location referenced by the `sval` argument to have
 38753 the value of the semaphore referenced by `sem` without affecting the state of the semaphore. The
 38754 updated value represents an actual semaphore value that occurred at some unspecified time
 38755 during the call, but it need not be the actual value of the semaphore when it is returned to the
 38756 calling process.

38757 If `sem` is locked, then the value returned by `sem_getvalue()` is either zero or a negative number
 38758 whose absolute value represents the number of processes waiting for the semaphore at some
 38759 unspecified time during the call.

38760 **RETURN VALUE**

38761 Upon successful completion, the `sem_getvalue()` function shall return a value of zero. Otherwise,
 38762 it shall return a value of -1 and set `errno` to indicate the error.

38763 **ERRORS**38764 The `sem_getvalue()` function shall fail if:38765 [EINVAL] The `sem` argument does not refer to a valid semaphore.38766 **EXAMPLES**

38767 None.

38768 **APPLICATION USAGE**

38769 The `sem_getvalue()` function is part of the Semaphores option and need not be available on all
 38770 implementations.

38771 **RATIONALE**

38772 None.

38773 **FUTURE DIRECTIONS**

38774 None.

38775 **SEE ALSO**

38776 `semctl()`, `semget()`, `semop()`, `sem_post()`, `sem_timedwait()`, `sem_trywait()`, `sem_wait()`, the Base
 38777 Definitions volume of IEEE Std 1003.1-2001, `<semaphore.h>`

38778 **CHANGE HISTORY**

38779 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38780 **Issue 6**38781 The `sem_getvalue()` function is marked as part of the Semaphores option.

38782 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 38783 implementation does not support the Semaphores option.

38784 The `sem_timedwait()` function is added to the SEE ALSO section for alignment with
 38785 IEEE Std 1003.1d-1999.

38786 The **restrict** keyword is added to the `sem_getvalue()` prototype for alignment with the
 38787 ISO/IEC 9899:1999 standard.

38788 **NAME**38789 sem_init — initialize an unnamed semaphore (**REALTIME**)38790 **SYNOPSIS**

38791 SEM #include <semaphore.h>

38792 int sem_init(sem_t *sem, int pshared, unsigned value);

38793

38794 **DESCRIPTION**

38795 The *sem_init()* function shall initialize the unnamed semaphore referred to by *sem*. The value of
 38796 the initialized semaphore shall be *value*. Following a successful call to *sem_init()*, the semaphore
 38797 may be used in subsequent calls to *sem_wait()*, *sem_trywait()*, *sem_post()*, and *sem_destroy()*.
 38798 This semaphore shall remain usable until the semaphore is destroyed.

38799 If the *pshared* argument has a non-zero value, then the semaphore is shared between processes;
 38800 in this case, any process that can access the semaphore *sem* can use *sem* for performing
 38801 *sem_wait()*, *sem_trywait()*, *sem_post()*, and *sem_destroy()* operations.

38802 Only *sem* itself may be used for performing synchronization. The result of referring to copies of
 38803 *sem* in calls to *sem_wait()*, *sem_trywait()*, *sem_post()*, and *sem_destroy()* is undefined.

38804 If the *pshared* argument is zero, then the semaphore is shared between threads of the process; any
 38805 thread in this process can use *sem* for performing *sem_wait()*, *sem_trywait()*, *sem_post()*, and
 38806 *sem_destroy()* operations. The use of the semaphore by threads other than those created in the
 38807 same process is undefined.

38808 Attempting to initialize an already initialized semaphore results in undefined behavior.

38809 **RETURN VALUE**

38810 Upon successful completion, the *sem_init()* function shall initialize the semaphore in *sem*.
 38811 Otherwise, it shall return -1 and set *errno* to indicate the error.

38812 **ERRORS**38813 The *sem_init()* function shall fail if:

38814 [EINVAL] The *value* argument exceeds {SEM_VALUE_MAX}.

38815 [ENOSPC] A resource required to initialize the semaphore has been exhausted, or the
 38816 limit on semaphores ({SEM_NSEMS_MAX}) has been reached.

38817 [EPERM] The process lacks the appropriate privileges to initialize the semaphore.

38818 **EXAMPLES**

38819 None.

38820 **APPLICATION USAGE**

38821 The *sem_init()* function is part of the Semaphores option and need not be available on all
 38822 implementations.

38823 **RATIONALE**

38824 Although this volume of IEEE Std 1003.1-2001 fails to specify a successful return value, it is
 38825 likely that a later version may require the implementation to return a value of zero if the call to
 38826 *sem_init()* is successful.

38827 **FUTURE DIRECTIONS**

38828 None.

38829 **SEE ALSO**

38830 *sem_destroy()*, *sem_post()*, *sem_timedwait()*, *sem_trywait()*, *sem_wait()*, the Base Definitions
38831 volume of IEEE Std 1003.1-2001, <semaphore.h>

38832 **CHANGE HISTORY**

38833 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38834 **Issue 6**

38835 The *sem_init()* function is marked as part of the Semaphores option.

38836 The [ENOSYS] error condition has been removed as stubs need not be provided if an
38837 implementation does not support the Semaphores option.

38838 The *sem_timedwait()* function is added to the SEE ALSO section for alignment with
38839 IEEE Std 1003.1d-1999.

38840 NAME

38841 sem_open — initialize and open a named semaphore (**REALTIME**)

38842 SYNOPSIS

38843 SEM

```
#include <semaphore.h>
```

38844

```
sem_t *sem_open(const char *name, int oflag, ...);
```

38845

38846 DESCRIPTION

38847 The *sem_open()* function shall establish a connection between a named semaphore and a process.
 38848 Following a call to *sem_open()* with semaphore name *name*, the process may reference the
 38849 semaphore associated with *name* using the address returned from the call. This semaphore may
 38850 be used in subsequent calls to *sem_wait()*, *sem_trywait()*, *sem_post()*, and *sem_close()*. The
 38851 semaphore remains usable by this process until the semaphore is closed by a successful call to
 38852 *sem_close()*, *_exit()*, or one of the *exec* functions.

38853 The *oflag* argument controls whether the semaphore is created or merely accessed by the call to
 38854 *sem_open()*. The following flag bits may be set in *oflag*:

38855 **O_CREAT** This flag is used to create a semaphore if it does not already exist. If **O_CREAT** is
 38856 set and the semaphore already exists, then **O_CREAT** has no effect, except as noted
 38857 under **O_EXCL**. Otherwise, *sem_open()* creates a named semaphore. The **O_CREAT**
 38858 flag requires a third and a fourth argument: *mode*, which is of type **mode_t**, and
 38859 *value*, which is of type **unsigned**. The semaphore is created with an initial value of
 38860 *value*. Valid initial values for semaphores are less than or equal to
 38861 {SEM_VALUE_MAX}.

38862 The user ID of the semaphore is set to the effective user ID of the process; the
 38863 group ID of the semaphore is set to a system default group ID or to the effective
 38864 group ID of the process. The permission bits of the semaphore are set to the value
 38865 of the *mode* argument except those set in the file mode creation mask of the
 38866 process. When bits in *mode* other than the file permission bits are specified, the
 38867 effect is unspecified.

38868 After the semaphore named *name* has been created by *sem_open()* with the
 38869 **O_CREAT** flag, other processes can connect to the semaphore by calling
 38870 *sem_open()* with the same value of *name*.

38871 **O_EXCL** If **O_EXCL** and **O_CREAT** are set, *sem_open()* fails if the semaphore *name* exists.
 38872 The check for the existence of the semaphore and the creation of the semaphore if
 38873 it does not exist are atomic with respect to other processes executing *sem_open()*
 38874 with **O_EXCL** and **O_CREAT** set. If **O_EXCL** is set and **O_CREAT** is not set, the
 38875 effect is undefined.

38876 If flags other than **O_CREAT** and **O_EXCL** are specified in the *oflag* parameter, the
 38877 effect is unspecified.

38878 The *name* argument points to a string naming a semaphore object. It is unspecified whether the
 38879 name appears in the file system and is visible to functions that take pathnames as arguments.
 38880 The *name* argument conforms to the construction rules for a pathname. If *name* begins with the
 38881 slash character, then processes calling *sem_open()* with the same value of *name* shall refer to the
 38882 same semaphore object, as long as that name has not been removed. If *name* does not begin with
 38883 the slash character, the effect is implementation-defined. The interpretation of slash characters
 38884 other than the leading slash character in *name* is implementation-defined.

38885 If a process makes multiple successful calls to *sem_open()* with the same value for *name*, the
 38886 same semaphore address shall be returned for each such successful call, provided that there

38887 have been no calls to *sem_unlink()* for this semaphore.

38888 References to copies of the semaphore produce undefined results.

38889 RETURN VALUE

38890 Upon successful completion, the *sem_open()* function shall return the address of the semaphore.
 38891 Otherwise, it shall return a value of SEM_FAILED and set *errno* to indicate the error. The symbol
 38892 SEM_FAILED is defined in the <semaphore.h> header. No successful return from *sem_open()*
 38893 shall return the value SEM_FAILED.

38894 ERRORS

38895 If any of the following conditions occur, the *sem_open()* function shall return SEM_FAILED and
 38896 set *errno* to the corresponding value:

38897 [EACCES] The named semaphore exists and the permissions specified by *oflag* are
 38898 denied, or the named semaphore does not exist and permission to create the
 38899 named semaphore is denied.

38900 [EEXIST] O_CREAT and O_EXCL are set and the named semaphore already exists.

38901 [EINTR] The *sem_open()* operation was interrupted by a signal.

38902 [EINVAL] The *sem_open()* operation is not supported for the given name, or O_CREAT
 38903 was specified in *oflag* and *value* was greater than {SEM_VALUE_MAX}.

38904 [EMFILE] Too many semaphore descriptors or file descriptors are currently in use by
 38905 this process.

38906 [ENAMETOOLONG]

38907 The length of the *name* argument exceeds {PATH_MAX} or a pathname
 38908 component is longer than {NAME_MAX}.

38909 [ENFILE] Too many semaphores are currently open in the system.

38910 [ENOENT] O_CREAT is not set and the named semaphore does not exist.

38911 [ENOSPC] There is insufficient space for the creation of the new named semaphore.

38912 EXAMPLES

38913 None.

38914 APPLICATION USAGE

38915 The *sem_open()* function is part of the Semaphores option and need not be available on all
 38916 implementations.

38917 RATIONALE

38918 Early drafts required an error return value of -1 with the type **sem_t *** for the *sem_open()*
 38919 function, which is not guaranteed to be portable across implementations. The revised text
 38920 provides the symbolic error code SEM_FAILED to eliminate the type conflict.

38921 FUTURE DIRECTIONS

38922 None.

38923 SEE ALSO

38924 *semctl()*, *semget()*, *semop()*, *sem_close()*, *sem_post()*, *sem_timedwait()*, *sem_trywait()*, *sem_unlink()*,
 38925 *sem_wait()*, the Base Definitions volume of IEEE Std 1003.1-2001, <semaphore.h>

38926 CHANGE HISTORY

38927 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38928 Issue 6

38929 The *sem_open()* function is marked as part of the Semaphores option.

38930 The [ENOSYS] error condition has been removed as stubs need not be provided if an
38931 implementation does not support the Semaphores option.

38932 The *sem_timedwait()* function is added to the SEE ALSO section for alignment with
38933 IEEE Std 1003.1d-1999.

38934 **NAME**38935 sem_post — unlock a semaphore (**REALTIME**)38936 **SYNOPSIS**

38937 SEM #include <semaphore.h>

38938 int sem_post(sem_t *sem);

38939

38940 **DESCRIPTION**38941 The *sem_post()* function shall unlock the semaphore referenced by *sem* by performing a
38942 semaphore unlock operation on that semaphore.38943 If the semaphore value resulting from this operation is positive, then no threads were blocked
38944 waiting for the semaphore to become unlocked; the semaphore value is simply incremented.38945 If the value of the semaphore resulting from this operation is zero, then one of the threads
38946 blocked waiting for the semaphore shall be allowed to return successfully from its call to
38947 *sem_wait()*. If the Process Scheduling option is supported, the thread to be unblocked shall be
38948 chosen in a manner appropriate to the scheduling policies and parameters in effect for the
38949 blocked threads. In the case of the schedulers SCHED_FIFO and SCHED_RR, the highest
38950 priority waiting thread shall be unblocked, and if there is more than one highest priority thread
38951 blocked waiting for the semaphore, then the highest priority thread that has been waiting the
38952 longest shall be unblocked. If the Process Scheduling option is not defined, the choice of a thread
38953 to unblock is unspecified.38954 SS If the Process Sporadic Server option is supported, and the scheduling policy is
38955 SCHED_SPORADIC, the semantics are as per SCHED_FIFO above.38956 The *sem_post()* function shall be reentrant with respect to signals and may be invoked from a
38957 signal-catching function.38958 **RETURN VALUE**38959 If successful, the *sem_post()* function shall return zero; otherwise, the function shall return -1
38960 and set *errno* to indicate the error.38961 **ERRORS**38962 The *sem_post()* function shall fail if:38963 [EINVAL] The *sem* argument does not refer to a valid semaphore.38964 **EXAMPLES**

38965 None.

38966 **APPLICATION USAGE**38967 The *sem_post()* function is part of the Semaphores option and need not be available on all
38968 implementations.38969 **RATIONALE**

38970 None.

38971 **FUTURE DIRECTIONS**

38972 None.

38973 **SEE ALSO**38974 *semctl()*, *semget()*, *semop()*, *sem_timedwait()*, *sem_trywait()*, *sem_wait()*, the Base Definitions
38975 volume of IEEE Std 1003.1-2001, <semaphore.h>

38976 CHANGE HISTORY

38977 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

38978 Issue 6

38979 The *sem_post()* function is marked as part of the Semaphores option.

38980 The [ENOSYS] error condition has been removed as stubs need not be provided if an
38981 implementation does not support the Semaphores option.

38982 The *sem_timedwait()* function is added to the SEE ALSO section for alignment with
38983 IEEE Std 1003.1d-1999.

38984 SCHED_SPORADIC is added to the list of scheduling policies for which the thread that is to be
38985 unblocked is specified for alignment with IEEE Std 1003.1d-1999.

38986 **NAME**38987 sem_timedwait — lock a semaphore (**ADVANCED REALTIME**)38988 **SYNOPSIS**

38989 SEM TMO #include <semaphore.h>

38990 #include <time.h>

```
38991     int sem_timedwait(sem_t *restrict sem,
38992                      const struct timespec *restrict abs_timeout);
```

38993

38994 **DESCRIPTION**

38995 The *sem_timedwait()* function shall lock the semaphore referenced by *sem* as in the *sem_wait()* function. However, if the semaphore cannot be locked without waiting for another process or thread to unlock the semaphore by performing a *sem_post()* function, this wait shall be terminated when the specified timeout expires.

38999 The timeout shall expire when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

39003 TMR If the Timers option is supported, the timeout shall be based on the **CLOCK_REALTIME** clock. If the Timers option is not supported, the timeout shall be based on the system clock as returned by the *time()* function. The resolution of the timeout shall be the resolution of the clock on which it is based. The **timespec** data type is defined as a structure in the **<time.h>** header.

39007 Under no circumstance shall the function fail with a timeout if the semaphore can be locked immediately. The validity of the *abs_timeout* need not be checked if the semaphore can be locked immediately.

39010 **RETURN VALUE**

39011 The *sem_timedwait()* function shall return zero if the calling process successfully performed the semaphore lock operation on the semaphore designated by *sem*. If the call was unsuccessful, the state of the semaphore shall be unchanged, and the function shall return a value of **-1** and set *errno* to indicate the error.

39015 **ERRORS**39016 The *sem_timedwait()* function shall fail if:

39017 [EINVAL] The *sem* argument does not refer to a valid semaphore.

39018 [EINVAL] The process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than zero or greater than or equal to 1 000 million.

39021 [ETIMEDOUT] The semaphore could not be locked before the specified timeout expired.

39022 The *sem_timedwait()* function may fail if:

39023 [EDEADLK] A deadlock condition was detected.

39024 [EINTR] A signal interrupted this function.

39025 EXAMPLES

39026 None.

39027 APPLICATION USAGE

39028 Applications using these functions may be subject to priority inversion, as discussed in the Base
39029 Definitions volume of IEEE Std 1003.1-2001, Section 3.285, Priority Inversion.

39030 The *sem_timedwait()* function is part of the Semaphores and Timeouts options and need not be
39031 provided on all implementations.

39032 RATIONALE

39033 None.

39034 FUTURE DIRECTIONS

39035 None.

39036 SEE ALSO

39037 *sem_post()*, *sem_trywait()*, *sem_wait()*, *semctl()*, *semget()*, *semop()*, *time()*, the Base Definitions
39038 volume of IEEE Std 1003.1-2001, <**semaphore.h**>, <**time.h**>

39039 CHANGE HISTORY

39040 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

39041 **NAME**39042 sem_trywait, sem_wait — lock a semaphore (**REALTIME**)39043 **SYNOPSIS**

39044 SEM #include <semaphore.h>

39045 int sem_trywait(sem_t *sem);

39046 int sem_wait(sem_t *sem);

39047

39048 **DESCRIPTION**

39049 The *sem_trywait()* function shall lock the semaphore referenced by *sem* only if the semaphore is
 39050 currently not locked; that is, if the semaphore value is currently positive. Otherwise, it shall not
 39051 lock the semaphore.

39052 The *sem_wait()* function shall lock the semaphore referenced by *sem* by performing a semaphore
 39053 lock operation on that semaphore. If the semaphore value is currently zero, then the calling
 39054 thread shall not return from the call to *sem_wait()* until it either locks the semaphore or the call is
 39055 interrupted by a signal.

39056 Upon successful return, the state of the semaphore shall be locked and shall remain locked until
 39057 the *sem_post()* function is executed and returns successfully.

39058 The *sem_wait()* function is interruptible by the delivery of a signal.

39059 **RETURN VALUE**

39060 The *sem_trywait()* and *sem_wait()* functions shall return zero if the calling process successfully
 39061 performed the semaphore lock operation on the semaphore designated by *sem*. If the call was
 39062 unsuccessful, the state of the semaphore shall be unchanged, and the function shall return a
 39063 value of -1 and set *errno* to indicate the error.

39064 **ERRORS**

39065 The *sem_trywait()* and *sem_wait()* functions shall fail if:

39066 [EAGAIN] The semaphore was already locked, so it cannot be immediately locked by the
 39067 *sem_trywait()* operation (*sem_trywait()* only).

39068 [EINVAL] The *sem* argument does not refer to a valid semaphore.

39069 The *sem_trywait()* and *sem_wait()* functions may fail if:

39070 [EDEADLK] A deadlock condition was detected.

39071 [EINTR] A signal interrupted this function.

39072 **EXAMPLES**

39073 None.

39074 **APPLICATION USAGE**

39075 Applications using these functions may be subject to priority inversion, as discussed in the Base
 39076 Definitions volume of IEEE Std 1003.1-2001, Section 3.285, Priority Inversion.

39077 The *sem_trywait()* and *sem_wait()* functions are part of the Semaphores option and need not be
 39078 provided on all implementations.

39079 **RATIONALE**

39080 None.

39081 FUTURE DIRECTIONS

39082 None.

39083 SEE ALSO

39084 *semctl()*, *semget()*, *semop()*, *sem_post()*, *sem_timedwait()*, the Base Definitions volume of
39085 IEEE Std 1003.1-2001, <**semaphore.h**>

39086 CHANGE HISTORY

39087 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

39088 Issue 6

39089 The *sem_trywait()* and *sem_wait()* functions are marked as part of the Semaphores option.

39090 The [ENOSYS] error condition has been removed as stubs need not be provided if an
39091 implementation does not support the Semaphores option.

39092 The *sem_timedwait()* function is added to the SEE ALSO section for alignment with
39093 IEEE Std 1003.1d-1999.

39094 **NAME**39095 sem_unlink — remove a named semaphore (**REALTIME**)39096 **SYNOPSIS**

39097 SEM #include <semaphore.h>

39098 int sem_unlink(const char *name);

39099

39100 **DESCRIPTION**

39101 The *sem_unlink()* function shall remove the semaphore named by the string *name*. If the
 39102 semaphore named by *name* is currently referenced by other processes, then *sem_unlink()* shall
 39103 have no effect on the state of the semaphore. If one or more processes have the semaphore open
 39104 when *sem_unlink()* is called, destruction of the semaphore is postponed until all references to the
 39105 semaphore have been destroyed by calls to *sem_close()*, *_exit()*, or *exec*. Calls to *sem_open()* to
 39106 recreate or reconnect to the semaphore refer to a new semaphore after *sem_unlink()* is called. The
 39107 *sem_unlink()* call shall not block until all references have been destroyed; it shall return
 39108 immediately.

39109 **RETURN VALUE**

39110 Upon successful completion, the *sem_unlink()* function shall return a value of 0. Otherwise, the
 39111 semaphore shall not be changed and the function shall return a value of -1 and set *errno* to
 39112 indicate the error.

39113 **ERRORS**39114 The *sem_unlink()* function shall fail if:

39115 [EACCES] Permission is denied to unlink the named semaphore.

39116 [ENAMETOOLONG]

39117 The length of the *name* argument exceeds {PATH_MAX} or a pathname
 39118 component is longer than {NAME_MAX}.

39119 [ENOENT] The named semaphore does not exist.

39120 **EXAMPLES**

39121 None.

39122 **APPLICATION USAGE**

39123 The *sem_unlink()* function is part of the Semaphores option and need not be available on all
 39124 implementations.

39125 **RATIONALE**

39126 None.

39127 **FUTURE DIRECTIONS**

39128 None.

39129 **SEE ALSO**

39130 *semctl()*, *semget()*, *semop()*, *sem_close()*, *sem_open()*, the Base Definitions volume of
 39131 IEEE Std 1003.1-2001, <semaphore.h>

39132 **CHANGE HISTORY**

39133 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

39134 **Issue 6**39135 The *sem_unlink()* function is marked as part of the Semaphores option.

39136 The [ENOSYS] error condition has been removed as stubs need not be provided if an
39137 implementation does not support the Semaphores option.

39138 **NAME**39139 sem_wait — lock a semaphore (**REALTIME**)39140 **SYNOPSIS**

39141 SEM #include <semaphore.h>

39142 int sem_wait(sem_t *sem);

39143

39144 **DESCRIPTION**39145 Refer to *sem_trywait()*.

39146 NAME

39147 semctl — XSI semaphore control operations

39148 SYNOPSIS

39149 XSI

```
#include <sys/sem.h>
```

39150

```
int semctl(int semid, int semnum, int cmd, ...);
```

39151

39152 DESCRIPTION

39153 The *semctl()* function operates on XSI semaphores (see the Base Definitions volume of
 39154 IEEE Std 1003.1-2001, Section 4.15, Semaphore). It is unspecified whether this function
 39155 interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on
 39156 page 41).

39157 The *semctl()* function provides a variety of semaphore control operations as specified by *cmd*.
 39158 The fourth argument is optional and depends upon the operation requested. If required, it is of
 39159 type **union semun**, which the application shall explicitly declare:

```
39160 union semun {
39161     int val;
39162     struct semid_ds *buf;
39163     unsigned short *array;
39164 } arg;
```

39165 The following semaphore control operations as specified by *cmd* are executed with respect to the
 39166 semaphore specified by *semid* and *semnum*. The level of permission required for each operation
 39167 is shown with each command; see Section 2.7 (on page 39). The symbolic names for the values
 39168 of *cmd* are defined in the **<sys/sem.h>** header:

39169	GETVAL	Return the value of <i>semval</i> ; see <sys/sem.h> . Requires read permission.
39170	SETVAL	Set the value of <i>semval</i> to <i>arg.val</i> , where <i>arg</i> is the value of the fourth argument to <i>semctl()</i> . When this command is successfully executed, the <i>semadj</i> value corresponding to the specified semaphore in all processes is cleared. Requires alter permission; see Section 2.7 (on page 39).

39174	GETPID	Return the value of <i>sempid</i> . Requires read permission.
-------	--------	---

39175	GETNCNT	Return the value of <i>semmcnt</i> . Requires read permission.
-------	---------	--

39176	GETZCNT	Return the value of <i>semzcnt</i> . Requires read permission.
-------	---------	--

39177 The following values of *cmd* operate on each *semval* in the set of semaphores:

39178	GETALL	Return the value of <i>semval</i> for each semaphore in the semaphore set and place into the array pointed to by <i>arg.array</i> , where <i>arg</i> is the fourth argument to <i>semctl()</i> . Requires read permission.
-------	--------	--

39181	SETALL	Set the value of <i>semval</i> for each semaphore in the semaphore set according to the array pointed to by <i>arg.array</i> , where <i>arg</i> is the fourth argument to <i>semctl()</i> . When this command is successfully executed, the <i>semadj</i> values corresponding to each specified semaphore in all processes are cleared. Requires alter permission.
-------	--------	---

39186 The following values of *cmd* are also available:

39187	IPC_STAT	Place the current value of each member of the semid_ds data structure associated with <i>semid</i> into the structure pointed to by <i>arg.buf</i> , where <i>arg</i> is the fourth argument to <i>semctl()</i> . The contents of this structure are defined in
-------	----------	--

39190		<sys/sem.h>. Requires read permission.
39191	IPC_SET	Set the value of the following members of the semid_ds data structure associated with <i>semid</i> to the corresponding value found in the structure pointed to by <i>arg.buf</i> , where <i>arg</i> is the fourth argument to <i>semctl()</i> :
39192		
39193		
39194		<i>sem_perm.uid</i>
39195		<i>sem_perm.gid</i>
39196		<i>sem_perm.mode</i>
39197		The mode bits specified in Section 2.7.1 (on page 40) are copied into the corresponding bits of the <i>sem_perm.mode</i> associated with <i>semid</i> . The stored values of any other bits are unspecified.
39198		
39199		
39200		This command can only be executed by a process that has an effective user ID equal to either that of a process with appropriate privileges or to the value of <i>sem_perm.cuid</i> or <i>sem_perm.uid</i> in the semid_ds data structure associated with <i>semid</i> .
39201		
39202		
39203		
39204	IPC_RMID	Remove the semaphore identifier specified by <i>semid</i> from the system and destroy the set of semaphores and semid_ds data structure associated with it. This command can only be executed by a process that has an effective user ID equal to either that of a process with appropriate privileges or to the value of <i>sem_perm.cuid</i> or <i>sem_perm.uid</i> in the semid_ds data structure associated with <i>semid</i> .
39205		
39206		
39207		
39208		
39209		
39210	RETURN VALUE	
39211		If successful, the value returned by <i>semctl()</i> depends on <i>cmd</i> as follows:
39212	GETVAL	The value of <i>semval</i> .
39213	GETPID	The value of <i>sempid</i> .
39214	GETNCNT	The value of <i>semmcnt</i> .
39215	GETZCNT	The value of <i>semzcnt</i> .
39216	All others	0.
39217		Otherwise, <i>semctl()</i> shall return -1 and set <i>errno</i> to indicate the error.
39218	ERRORS	
39219		The <i>semctl()</i> function shall fail if:
39220	[EACCES]	Operation permission is denied to the calling process; see Section 2.7 (on page 39).
39221		
39222	[EINVAL]	The value of <i>semid</i> is not a valid semaphore identifier, or the value of <i>semnum</i> is less than 0 or greater than or equal to <i>sem_nsems</i> , or the value of <i>cmd</i> is not a valid command.
39223		
39224		
39225	[EPERM]	The argument <i>cmd</i> is equal to IPC_RMID or IPC_SET and the effective user ID of the calling process is not equal to that of a process with appropriate privileges and it is not equal to the value of <i>sem_perm.cuid</i> or <i>sem_perm.uid</i> in the data structure associated with <i>semid</i> .
39226		
39227		
39228		
39229	[ERANGE]	The argument <i>cmd</i> is equal to SETVAL or SETALL and the value to which <i>semval</i> is to be set is greater than the system-imposed maximum.
39230		

39231 **EXAMPLES**

39232 None.

39233 **APPLICATION USAGE**

39234 The fourth parameter in the SYNOPSIS section is now specified as " . . . " in order to avoid a
39235 clash with the ISO C standard when referring to the union *semun* (as defined in Issue 3) and for
39236 backwards-compatibility.

39237 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
39238 Application developers who need to use IPC should design their applications so that modules
39239 using the IPC routines described in Section 2.7 (on page 39) can be easily modified to use the
39240 alternative interfaces.

39241 **RATIONALE**

39242 None.

39243 **FUTURE DIRECTIONS**

39244 None.

39245 **SEE ALSO**

39246 Section 2.7 (on page 39), Section 2.8 (on page 41), *semget()*, *semop()*, *sem_close()*, *sem_destroy()*,
39247 *sem_getvalue()*, *sem_init()*, *sem_open()*, *sem_post()*, *sem_unlink()*, *sem_wait()*, the Base Definitions
39248 volume of IEEE Std 1003.1-2001, <sys/sem.h>

39249 **CHANGE HISTORY**

39250 First released in Issue 2. Derived from Issue 2 of the SVID.

39251 **Issue 5**

39252 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
39253 DIRECTIONS to the APPLICATION USAGE section.

39254 NAME

39255 semget — get set of XSI semaphores

39256 SYNOPSIS

39257 XSI

```
#include <sys/sem.h>
```

39258

```
int semget(key_t key, int nsems, int semflg);
```

39259

39260 DESCRIPTION

39261 The *semget()* function operates on XSI semaphores (see the Base Definitions volume of
 39262 IEEE Std 1003.1-2001, Section 4.15, Semaphore). It is unspecified whether this function
 39263 interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on
 39264 page 41).

39265 The *semget()* function shall return the semaphore identifier associated with *key*.

39266 A semaphore identifier with its associated **semid_ds** data structure and its associated set of
 39267 *nsems* semaphores (see <sys/sem.h>) is created for *key* if one of the following is true:

- 39268 • The argument *key* is equal to IPC_PRIVATE.
- 39269 • The argument *key* does not already have a semaphore identifier associated with it and (*semflg*
 39270 &IPC_CREAT) is non-zero.

39271 Upon creation, the **semid_ds** data structure associated with the new semaphore identifier is
 39272 initialized as follows:

- 39273 • In the operation permissions structure *sem_perm.cuid*, *sem_perm.uid*, *sem_perm.cgid*, and
 39274 *sem_perm.gid* shall be set equal to the effective user ID and effective group ID, respectively, of
 39275 the calling process.
- 39276 • The low-order 9 bits of *sem_perm.mode* shall be set equal to the low-order 9 bits of *semflg*.
- 39277 • The variable *sem_nsems* shall be set equal to the value of *nsems*.
- 39278 • The variable *sem_otime* shall be set equal to 0 and *sem_ctime* shall be set equal to the current
 39279 time.
- 39280 • The data structure associated with each semaphore in the set shall not be initialized. The
 39281 *semctl()* function with the command SETVAL or SETALL can be used to initialize each
 39282 semaphore.

39283 RETURN VALUE

39284 Upon successful completion, *semget()* shall return a non-negative integer, namely a semaphore
 39285 identifier; otherwise, it shall return -1 and set *errno* to indicate the error.

39286 ERRORS

39287 The *semget()* function shall fail if:

- | | | |
|-------|----------|---|
| 39288 | [EACCES] | A semaphore identifier exists for <i>key</i> , but operation permission as specified by the low-order 9 bits of <i>semflg</i> would not be granted; see Section 2.7 (on page 39). |
| 39289 | | |
| 39290 | | |
| 39291 | [EEXIST] | A semaphore identifier exists for the argument <i>key</i> but ((<i>semflg</i> &IPC_CREAT) &&(<i>semflg</i> &IPC_EXCL)) is non-zero. |
| 39292 | | |
| 39293 | [EINVAL] | The value of <i>nsems</i> is either less than or equal to 0 or greater than the system-imposed limit, or a semaphore identifier exists for the argument <i>key</i> , but the number of semaphores in the set associated with it is less than <i>nsems</i> and <i>nsems</i> is not equal to 0. |
| 39294 | | |
| 39295 | | |
| 39296 | | |

39297 [ENOENT] A semaphore identifier does not exist for the argument *key* and (*semflg*
 39298 &IPC_CREAT) is equal to 0.

39299 [ENOSPC] A semaphore identifier is to be created but the system-imposed limit on the
 39300 maximum number of allowed semaphores system-wide would be exceeded.

39301 EXAMPLES

39302 Creating a Semaphore Identifier

39303 The following example gets a unique semaphore key using the *ftok()* function, then gets a
 39304 semaphore ID associated with that key using the *semget()* function (the first call also tests to
 39305 make sure the semaphore exists). If the semaphore does not exist, the program creates it, as
 39306 shown by the second call to *semget()*. In creating the semaphore for the queuing process, the
 39307 program attempts to create one semaphore with read/write permission for all. It also uses the
 39308 IPC_EXCL flag, which forces *semget()* to fail if the semaphore already exists.

39309 After creating the semaphore, the program uses a call to *semop()* to initialize it to the values in
 39310 the *sbuf* array. The number of processes that can execute concurrently without queuing is
 39311 initially set to 2. The final call to *semget()* creates a semaphore identifier that can be used later in
 39312 the program.

```

39313 #include <sys/types.h>
39314 #include <stdio.h>
39315 #include <sys/ipc.h>
39316 #include <sys/sem.h>
39317 #include <sys/stat.h>
39318 #include <errno.h>
39319 #include <unistd.h>
39320 #include <stdlib.h>
39321 #include <pwd.h>
39322 #include <fcntl.h>
39323 #include <limits.h>
39324 ...
39325 key_t semkey;
39326 int semid, pfd, fv;
39327 struct sembuf sbuf;
39328 char *lgn;
39329 char filename[PATH_MAX+1];
39330 struct stat outstat;
39331 struct passwd *pw;
39332 ...
39333 /* Get unique key for semaphore. */
39334 if ((semkey = ftok("/tmp", 'a')) == (key_t) -1) {
39335     perror("IPC error: ftok"); exit(1);
39336 }
39337
39338 /* Get semaphore ID associated with this key. */
39339 if ((semid = semget(semkey, 0, 0)) == -1) {
39340     /* Semaphore does not exist - Create. */
39341     if ((semid = semget(semkey, 1, IPC_CREAT | IPC_EXCL | S_IRUSR |
39342         S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH)) != -1)
39343     {
39344         /* Initialize the semaphore. */
39345         sbuf.sem_num = 0;
  
```



```

39345         sbuf.sem_op = 2; /* This is the number of runs
39346                        without queuing. */
39347         sbuf.sem_flg = 0;
39348         if (semop(semid, &sbuf, 1) == -1) {
39349             perror("IPC error: semop"); exit(1);
39350         }
39351     }
39352     else if (errno == EEXIST) {
39353         if ((semid = semget(semkey, 0, 0)) == -1) {
39354             perror("IPC error 1: semget"); exit(1);
39355         }
39356     }
39357     else {
39358         perror("IPC error 2: semget"); exit(1);
39359     }
39360 }
39361 ...

```

39362 APPLICATION USAGE

39363 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
 39364 Application developers who need to use IPC should design their applications so that modules
 39365 using the IPC routines described in Section 2.7 (on page 39) can be easily modified to use the
 39366 alternative interfaces.

39367 RATIONALE

39368 None.

39369 FUTURE DIRECTIONS

39370 None.

39371 SEE ALSO

39372 Section 2.7 (on page 39), Section 2.8 (on page 41), *semctl()*, *semop()*, *sem_close()*, *sem_destroy()*,
 39373 *sem_getvalue()*, *sem_init()*, *sem_open()*, *sem_post()*, *sem_unlink()*, *sem_wait()*, the Base Definitions
 39374 volume of IEEE Std 1003.1-2001, <sys/sem.h>

39375 CHANGE HISTORY

39376 First released in Issue 2. Derived from Issue 2 of the SVID.

39377 Issue 5

39378 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
 39379 DIRECTIONS to a new APPLICATION USAGE section.

39380 NAME

39381 semop — XSI semaphore operations

39382 SYNOPSIS

39383 XSI

```
#include <sys/sem.h>
```

39384

```
int semop(int semid, struct sembuf *sops, size_t nsops);
```

39385

39386 DESCRIPTION

39387 The *semop()* function operates on XSI semaphores (see the Base Definitions volume of
 39388 IEEE Std 1003.1-2001, Section 4.15, Semaphore). It is unspecified whether this function
 39389 interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on
 39390 page 41).

39391 The *semop()* function shall perform atomically a user-defined array of semaphore operations on
 39392 the set of semaphores associated with the semaphore identifier specified by the argument *semid*.

39393 The argument *sops* is a pointer to a user-defined array of semaphore operation structures. The
 39394 implementation shall not modify elements of this array unless the application uses
 39395 implementation-defined extensions.

39396 The argument *nsops* is the number of such structures in the array.

39397 Each structure, **sembuf**, includes the following members:

39398

39399

39400

39401

39402

Member Type	Member Name	Description
short	<i>sem_num</i>	Semaphore number.
short	<i>sem_op</i>	Semaphore operation.
short	<i>sem_flg</i>	Operation flags.

39403 Each semaphore operation specified by *sem_op* is performed on the corresponding semaphore
 39404 specified by *semid* and *sem_num*.

39405 The variable *sem_op* specifies one of three semaphore operations:

39406 1. If *sem_op* is a negative integer and the calling process has alter permission, one of the
 39407 following shall occur:

39408 • If *semval* (see <sys/sem.h>) is greater than or equal to the absolute value of *sem_op*, the
 39409 absolute value of *sem_op* is subtracted from *semval*. Also, if (*sem_flg* & SEM_UNDO) is
 39410 non-zero, the absolute value of *sem_op* shall be added to the calling process' *semadj*
 39411 value for the specified semaphore.

39412 • If *semval* is less than the absolute value of *sem_op* and (*sem_flg* & IPC_NOWAIT) is non-
 39413 zero, *semop()* shall return immediately.

39414 • If *semval* is less than the absolute value of *sem_op* and (*sem_flg* & IPC_NOWAIT) is 0,
 39415 *semop()* shall increment the *semncnt* associated with the specified semaphore and
 39416 suspend execution of the calling thread until one of the following conditions occurs:

39417 — The value of *semval* becomes greater than or equal to the absolute value of *sem_op*.
 39418 When this occurs, the value of *semncnt* associated with the specified semaphore
 39419 shall be decremented, the absolute value of *sem_op* shall be subtracted from *semval*
 39420 and, if (*sem_flg* & SEM_UNDO) is non-zero, the absolute value of *sem_op* shall be
 39421 added to the calling process' *semadj* value for the specified semaphore.

39422 — The *semid* for which the calling thread is awaiting action is removed from the
 39423 system. When this occurs, *errno* shall be set equal to [EIDRM] and -1 shall be

39424 returned.

39425 — The calling thread receives a signal that is to be caught. When this occurs, the value
39426 of *semncnt* associated with the specified semaphore shall be decremented, and the
39427 calling thread shall resume execution in the manner prescribed in *sigaction()*.

39428 2. If *sem_op* is a positive integer and the calling process has alter permission, the value of
39429 *sem_op* shall be added to *semval* and, if (*sem_flg* & SEM_UNDO) is non-zero, the value of
39430 *sem_op* shall be subtracted from the calling process' *semadj* value for the specified
39431 semaphore.

39432 3. If *sem_op* is 0 and the calling process has read permission, one of the following shall occur:

39433 • If *semval* is 0, *semop()* shall return immediately.

39434 • If *semval* is non-zero and (*sem_flg* & IPC_NOWAIT) is non-zero, *semop()* shall return
39435 immediately.

39436 • If *semval* is non-zero and (*sem_flg* & IPC_NOWAIT) is 0, *semop()* shall increment the
39437 *semzcnt* associated with the specified semaphore and suspend execution of the calling
39438 thread until one of the following occurs:

39439 — The value of *semval* becomes 0, at which time the value of *semzcnt* associated with
39440 the specified semaphore shall be decremented.

39441 — The *semid* for which the calling thread is awaiting action is removed from the
39442 system. When this occurs, *errno* shall be set equal to [EIDRM] and -1 shall be
39443 returned.

39444 — The calling thread receives a signal that is to be caught. When this occurs, the value
39445 of *semzcnt* associated with the specified semaphore shall be decremented, and the
39446 calling thread shall resume execution in the manner prescribed in *sigaction()*.

39447 Upon successful completion, the value of *sempid* for each semaphore specified in the array
39448 pointed to by *sops* shall be set equal to the process ID of the calling process.

39449 **RETURN VALUE**

39450 Upon successful completion, *semop()* shall return 0; otherwise, it shall return -1 and set *errno* to
39451 indicate the error.

39452 **ERRORS**

39453 The *semop()* function shall fail if:

39454 [E2BIG] The value of *nsops* is greater than the system-imposed maximum.

39455 [EACCES] Operation permission is denied to the calling process; see Section 2.7 (on page
39456 39).

39457 [EAGAIN] The operation would result in suspension of the calling process but (*sem_flg*
39458 & IPC_NOWAIT) is non-zero.

39459 [EFBIG] The value of *sem_num* is less than 0 or greater than or equal to the number of
39460 semaphores in the set associated with *semid*.

39461 [EIDRM] The semaphore identifier *semid* is removed from the system.

39462 [EINTR] The *semop()* function was interrupted by a signal.

39463 [EINVAL] The value of *semid* is not a valid semaphore identifier, or the number of
39464 individual semaphores for which the calling process requests a SEM_UNDO
39465 would exceed the system-imposed limit.

39466 [ENOSPC] The limit on the number of individual processes requesting a SEM_UNDO
 39467 would be exceeded.

39468 [ERANGE] An operation would cause a *semval* to overflow the system-imposed limit, or
 39469 an operation would cause a *semadj* value to overflow the system-imposed
 39470 limit.

39471 EXAMPLES

39472 Setting Values in Semaphores

39473 The following example sets the values of the two semaphores associated with the *semid*
 39474 identifier to the values contained in the *sb* array.

```
39475 #include <sys/sem.h>
39476 ...
39477 int semid;
39478 struct sembuf sb[2];
39479 int nsops = 2;
39480 int result;

39481 /* Adjust value of semaphore in the semaphore array semid. */
39482 sb[0].sem_num = 0;
39483 sb[0].sem_op = -1;
39484 sb[0].sem_flg = SEM_UNDO | IPC_NOWAIT;
39485 sb[1].sem_num = 1;
39486 sb[1].sem_op = 1;
39487 sb[1].sem_flg = 0;

39488 result = semop(semid, sb, nsops);
```

39489 Creating a Semaphore Identifier

39490 The following example gets a unique semaphore key using the *ftok()* function, then gets a
 39491 semaphore ID associated with that key using the *semget()* function (the first call also tests to
 39492 make sure the semaphore exists). If the semaphore does not exist, the program creates it, as
 39493 shown by the second call to *semget()*. In creating the semaphore for the queuing process, the
 39494 program attempts to create one semaphore with read/write permission for all. It also uses the
 39495 IPC_EXCL flag, which forces *semget()* to fail if the semaphore already exists.

39496 After creating the semaphore, the program uses a call to *semop()* to initialize it to the values in
 39497 the *sbuf* array. The number of processes that can execute concurrently without queuing is
 39498 initially set to 2. The final call to *semget()* creates a semaphore identifier that can be used later in
 39499 the program.

39500 The final call to *semop()* acquires the semaphore and waits until it is free; the SEM_UNDO
 39501 option releases the semaphore when the process exits, waiting until there are less than two
 39502 processes running concurrently.

```
39503 #include <sys/types.h>
39504 #include <stdio.h>
39505 #include <sys/ipc.h>
39506 #include <sys/sem.h>
39507 #include <sys/stat.h>
39508 #include <errno.h>
39509 #include <unistd.h>
39510 #include <stdlib.h>
```



```

39511     #include <pwd.h>
39512     #include <fcntl.h>
39513     #include <limits.h>
39514     ...
39515     key_t semkey;
39516     int semid, pfd, fv;
39517     struct sembuf sbuf;
39518     char *lgn;
39519     char filename[PATH_MAX+1];
39520     struct stat outstat;
39521     struct passwd *pw;
39522     ...
39523     /* Get unique key for semaphore. */
39524     if ((semkey = ftok("/tmp", 'a')) == (key_t) -1) {
39525         perror("IPC error: ftok"); exit(1);
39526     }
39527     /* Get semaphore ID associated with this key. */
39528     if ((semid = semget(semkey, 0, 0)) == -1) {
39529         /* Semaphore does not exist - Create. */
39530         if ((semid = semget(semkey, 1, IPC_CREAT | IPC_EXCL | S_IRUSR |
39531             S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH)) != -1)
39532         {
39533             /* Initialize the semaphore. */
39534             sbuf.sem_num = 0;
39535             sbuf.sem_op = 2; /* This is the number of runs without queuing. */
39536             sbuf.sem_flg = 0;
39537             if (semop(semid, &sbuf, 1) == -1) {
39538                 perror("IPC error: semop"); exit(1);
39539             }
39540         }
39541         else if (errno == EEXIST) {
39542             if ((semid = semget(semkey, 0, 0)) == -1) {
39543                 perror("IPC error 1: semget"); exit(1);
39544             }
39545         }
39546         else {
39547             perror("IPC error 2: semget"); exit(1);
39548         }
39549     }
39550     ...
39551     sbuf.sem_num = 0;
39552     sbuf.sem_op = -1;
39553     sbuf.sem_flg = SEM_UNDO;
39554     if (semop(semid, &sbuf, 1) == -1) {
39555         perror("IPC Error: semop"); exit(1);
39556     }

```

39557 APPLICATION USAGE

39558 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
39559 Application developers who need to use IPC should design their applications so that modules
39560 using the IPC routines described in Section 2.7 (on page 39) can be easily modified to use the
39561 alternative interfaces.

39562 **RATIONALE**

39563 None.

39564 **FUTURE DIRECTIONS**

39565 None.

39566 **SEE ALSO**

39567 Section 2.7 (on page 39), Section 2.8 (on page 41), *exec*, *exit()*, *fork()*, *semctl()*, *semget()*,
39568 *sem_close()*, *sem_destroy()*, *sem_getvalue()*, *sem_init()*, *sem_open()*, *sem_post()*, *sem_unlink()*,
39569 *sem_wait()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<sys/ipc.h>**, **<sys/sem.h>**,
39570 **<sys/types.h>**

39571 **CHANGE HISTORY**

39572 First released in Issue 2. Derived from Issue 2 of the SVID.

39573 **Issue 5**

39574 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
39575 DIRECTIONS to a new APPLICATION USAGE section.

39576 **NAME**39577 **send** — send a message on a socket39578 **SYNOPSIS**

39579 #include <sys/socket.h>

39580 ssize_t send(int *socket*, const void **buffer*, size_t *length*, int *flags*);39581 **DESCRIPTION**39582 The *send()* function shall initiate transmission of a message from the specified socket to its peer.39583 The *send()* function shall send a message only when the socket is connected (including when the
39584 peer of a connectionless socket has been set via *connect()*).39585 The *send()* function takes the following arguments:39586 *socket* Specifies the socket file descriptor.39587 *buffer* Points to the buffer containing the message to send.39588 *length* Specifies the length of the message in bytes.39589 *flags* Specifies the type of message transmission. Values of this argument are
39590 formed by logically OR'ing zero or more of the following flags:

39591 MSG_EOR Terminates a record (if supported by the protocol).

39592 MSG_OOB Sends out-of-band data on sockets that support out-of-band
39593 communications. The significance and semantics of out-of-
39594 band data are protocol-specific.39595 The length of the message to be sent is specified by the *length* argument. If the message is too
39596 long to pass through the underlying protocol, *send()* shall fail and no data shall be transmitted.39597 Successful completion of a call to *send()* does not guarantee delivery of the message. A return
39598 value of -1 indicates only locally-detected errors.39599 If space is not available at the sending socket to hold the message to be transmitted, and the
39600 socket file descriptor does not have O_NONBLOCK set, *send()* shall block until space is
39601 available. If space is not available at the sending socket to hold the message to be transmitted,
39602 and the socket file descriptor does have O_NONBLOCK set, *send()* shall fail. The *select()* and
39603 *poll()* functions can be used to determine when it is possible to send more data.39604 The socket in use may require the process to have appropriate privileges to use the *send()*
39605 function.39606 **RETURN VALUE**39607 Upon successful completion, *send()* shall return the number of bytes sent. Otherwise, -1 shall be
39608 returned and *errno* set to indicate the error.39609 **ERRORS**39610 The *send()* function shall fail if:

39611 [EAGAIN] or [EWOULDBLOCK]

39612 The socket's file descriptor is marked O_NONBLOCK and the requested
39613 operation would block.39614 [EBADF] The *socket* argument is not a valid file descriptor.

39615 [ECONNRESET] A connection was forcibly closed by a peer.

39616 [EDESTADDRREQ]

39617 The socket is not connection-mode and no peer address is set.

39618	[EINTR]	A signal interrupted <i>send()</i> before any data was transmitted.
39619	[EMSGSIZE]	The message is too large to be sent all at once, as the socket requires.
39620	[ENOTCONN]	The socket is not connected or otherwise has not had the peer pre-specified.
39621	[ENOTSOCK]	The <i>socket</i> argument does not refer to a socket.
39622	[EOPNOTSUPP]	The <i>socket</i> argument is associated with a socket that does not support one or more of the values set in <i>flags</i> .
39623		
39624	[EPIPE]	The socket is shut down for writing, or the socket is connection-mode and is no longer connected. In the latter case, and if the socket is of type <code>SOCK_STREAM</code> , the <code>SIGPIPE</code> signal is generated to the calling thread.
39625		
39626		
39627		The <i>send()</i> function may fail if:
39628	[EACCES]	The calling process does not have the appropriate privileges.
39629	[EIO]	An I/O error occurred while reading from or writing to the file system.
39630	[ENETDOWN]	The local network interface used to reach the destination is down.
39631	[ENETUNREACH]	
39632		No route to the network is present.
39633	[ENOBUFS]	Insufficient resources were available in the system to perform the operation.

39634 EXAMPLES

39635 None.

39636 APPLICATION USAGE

39637 The *send()* function is equivalent to *sendto()* with a null pointer *dest_len* argument, and to *write()*
 39638 if no flags are used.

39639 RATIONALE

39640 None.

39641 FUTURE DIRECTIONS

39642 None.

39643 SEE ALSO

39644 *connect()*, *getsockopt()*, *poll()*, *recv()*, *recvfrom()*, *recvmsg()*, *select()*, *sendmsg()*, *sendto()*,
 39645 *setsockopt()*, *shutdown()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-2001,
 39646 <sys/socket.h>

39647 CHANGE HISTORY

39648 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

39649 **NAME**39650 `sendmsg` — send a message on a socket using a message structure39651 **SYNOPSIS**39652 `#include <sys/socket.h>`39653 `ssize_t sendmsg(int socket, const struct msghdr *message, int flags);`39654 **DESCRIPTION**

39655 The `sendmsg()` function shall send a message through a connection-mode or connectionless-mode socket. If the socket is connectionless-mode, the message shall be sent to the address specified by **msghdr**. If the socket is connection-mode, the destination address in **msghdr** shall be ignored.

39659 The `sendmsg()` function takes the following arguments:

39660	<i>socket</i>	Specifies the socket file descriptor.
39661	<i>message</i>	Points to a msghdr structure, containing both the destination address and the buffers for the outgoing message. The length and format of the address depend on the address family of the socket. The <i>msg_flags</i> member is ignored.
39662		
39663		
39664	<i>flags</i>	Specifies the type of message transmission. The application may specify 0 or the following flag:
39665		
39666	MSG_EOR	Terminates a record (if supported by the protocol).
39667	MSG_OOB	Sends out-of-band data on sockets that support out-of-bound data. The significance and semantics of out-of-band data are protocol-specific.
39668		
39669		

39670 The *msg_iov* and *msg_iovlen* fields of *message* specify zero or more buffers containing the data to be sent. *msg_iov* points to an array of **iovec** structures; *msg_iovlen* shall be set to the dimension of this array. In each **iovec** structure, the *iov_base* field specifies a storage area and the *iov_len* field gives its size in bytes. Some of these sizes can be zero. The data from each storage area indicated by *msg_iov* is sent in turn.

39675 Successful completion of a call to `sendmsg()` does not guarantee delivery of the message. A return value of `-1` indicates only locally-detected errors.

39677 If space is not available at the sending socket to hold the message to be transmitted and the socket file descriptor does not have `O_NONBLOCK` set, the `sendmsg()` function shall block until space is available. If space is not available at the sending socket to hold the message to be transmitted and the socket file descriptor does have `O_NONBLOCK` set, the `sendmsg()` function shall fail.

39682 If the socket protocol supports broadcast and the specified address is a broadcast address for the socket protocol, `sendmsg()` shall fail if the `SO_BROADCAST` option is not set for the socket.

39684 The socket in use may require the process to have appropriate privileges to use the `sendmsg()` function.

39686 **RETURN VALUE**

39687 Upon successful completion, `sendmsg()` shall return the number of bytes sent. Otherwise, `-1` shall be returned and *errno* set to indicate the error.

39689 **ERRORS**39690 The `sendmsg()` function shall fail if:39691 `[EAGAIN]` or `[EWOULDBLOCK]`39692 The socket's file descriptor is marked `O_NONBLOCK` and the requested

39693		operation would block.
39694	[EAFNOSUPPORT]	
39695		Addresses in the specified address family cannot be used with this socket.
39696	[EBADF]	The <i>socket</i> argument is not a valid file descriptor.
39697	[ECONNRESET]	A connection was forcibly closed by a peer.
39698	[EINTR]	A signal interrupted <i>sendmsg()</i> before any data was transmitted.
39699	[EINVAL]	The sum of the <i>iov_len</i> values overflows an ssize_t .
39700	[EMSGSIZE]	The message is too large to be sent all at once (as the socket requires), or the <i>msg_iovlen</i> member of the msghdr structure pointed to by <i>message</i> is less than or equal to 0 or is greater than {IOV_MAX}.
39701		
39702		
39703	[ENOTCONN]	The socket is connection-mode but is not connected.
39704	[ENOTSOCK]	The <i>socket</i> argument does not refer to a socket.
39705	[EOPNOTSUPP]	The <i>socket</i> argument is associated with a socket that does not support one or more of the values set in <i>flags</i> .
39706		
39707	[EPIPE]	The socket is shut down for writing, or the socket is connection-mode and is no longer connected. In the latter case, and if the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling thread.
39708		
39709		
39710		If the address family of the socket is AF_UNIX, then <i>sendmsg()</i> shall fail if:
39711	[EIO]	An I/O error occurred while reading from or writing to the file system.
39712	[ELOOP]	A loop exists in symbolic links encountered during resolution of the pathname in the socket address.
39713		
39714	[ENAMETOOLONG]	
39715		A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.
39716		
39717	[ENOENT]	A component of the pathname does not name an existing file or the path name is an empty string.
39718		
39719	[ENOTDIR]	A component of the path prefix of the pathname in the socket address is not a directory.
39720		
39721		The <i>sendmsg()</i> function may fail if:
39722	[EACCES]	Search permission is denied for a component of the path prefix; or write access to the named socket is denied.
39723		
39724	[EDESTADDRREQ]	
39725		The socket is not connection-mode and does not have its peer address set, and no destination address was specified.
39726		
39727	[EHOSTUNREACH]	
39728		The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).
39729		
39730	[EIO]	An I/O error occurred while reading from or writing to the file system.
39731	[EISCONN]	A destination address was specified and the socket is already connected.
39732	[ENETDOWN]	The local network interface used to reach the destination is down.

- 39733 [ENETUNREACH]
39734 No route to the network is present.
- 39735 [ENOBUFS] Insufficient resources were available in the system to perform the operation.
- 39736 [ENOMEM] Insufficient memory was available to fulfill the request.
- 39737 If the address family of the socket is AF_UNIX, then *sendmsg()* may fail if:
- 39738 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
39739 resolution of the pathname in the socket address.
- 39740 [ENAMETOOLONG]
39741 Pathname resolution of a symbolic link produced an intermediate result
39742 whose length exceeds {PATH_MAX}.
- 39743 **EXAMPLES**
39744 Done.
- 39745 **APPLICATION USAGE**
39746 The *select()* and *poll()* functions can be used to determine when it is possible to send more data.
- 39747 **RATIONALE**
39748 None.
- 39749 **FUTURE DIRECTIONS**
39750 None.
- 39751 **SEE ALSO**
39752 *getsockopt()*, *poll()*, *recv()*, *recvfrom()*, *recvmsg()*, *select()*, *send()*, *sendto()*, *setsockopt()*,
39753 *shutdown()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>
- 39754 **CHANGE HISTORY**
39755 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
- 39756 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
39757 [ELOOP] error condition is added.

39758 NAME

39759 sendto — send a message on a socket

39760 SYNOPSIS

39761 #include <sys/socket.h>

39762 ssize_t sendto(int socket, const void *message, size_t length,

39763 int flags, const struct sockaddr *dest_addr,

39764 socklen_t dest_len);

39765 DESCRIPTION

39766 The *sendto()* function shall send a message through a connection-mode or connectionless-mode
 39767 socket. If the socket is connectionless-mode, the message shall be sent to the address specified by
 39768 *dest_addr*. If the socket is connection-mode, *dest_addr* shall be ignored.

39769 The *sendto()* function takes the following arguments:

39770 *socket* Specifies the socket file descriptor.

39771 *message* Points to a buffer containing the message to be sent.

39772 *length* Specifies the size of the message in bytes.

39773 *flags* Specifies the type of message transmission. Values of this argument are
 39774 formed by logically OR'ing zero or more of the following flags:

39775 MSG_EOR Terminates a record (if supported by the protocol).

39776 MSG_OOB Sends out-of-band data on sockets that support out-of-band
 39777 data. The significance and semantics of out-of-band data are
 39778 protocol-specific.

39779 *dest_addr* Points to a **sockaddr** structure containing the destination address. The length
 39780 and format of the address depend on the address family of the socket.

39781 *dest_len* Specifies the length of the **sockaddr** structure pointed to by the *dest_addr*
 39782 argument.

39783 If the socket protocol supports broadcast and the specified address is a broadcast address for the
 39784 socket protocol, *sendto()* shall fail if the SO_BROADCAST option is not set for the socket.

39785 The *dest_addr* argument specifies the address of the target. The *length* argument specifies the
 39786 length of the message.

39787 Successful completion of a call to *sendto()* does not guarantee delivery of the message. A return
 39788 value of -1 indicates only locally-detected errors.

39789 If space is not available at the sending socket to hold the message to be transmitted and the
 39790 socket file descriptor does not have O_NONBLOCK set, *sendto()* shall block until space is
 39791 available. If space is not available at the sending socket to hold the message to be transmitted
 39792 and the socket file descriptor does have O_NONBLOCK set, *sendto()* shall fail.

39793 The socket in use may require the process to have appropriate privileges to use the *sendto()*
 39794 function.

39795 RETURN VALUE

39796 Upon successful completion, *sendto()* shall return the number of bytes sent. Otherwise, -1 shall
 39797 be returned and *errno* set to indicate the error.

39798 **ERRORS**

39799	The <i>sendto()</i> function shall fail if:	
39800	[EAFNOSUPPORT]	
39801		Addresses in the specified address family cannot be used with this socket.
39802	[EAGAIN] or [EWOULDBLOCK]	
39803		The socket's file descriptor is marked O_NONBLOCK and the requested
39804		operation would block.
39805	[EBADF]	The <i>socket</i> argument is not a valid file descriptor.
39806	[ECONNRESET]	A connection was forcibly closed by a peer.
39807	[EINTR]	A signal interrupted <i>sendto()</i> before any data was transmitted.
39808	[EMSGSIZE]	The message is too large to be sent all at once, as the socket requires.
39809	[ENOTCONN]	The socket is connection-mode but is not connected.
39810	[ENOTSOCK]	The <i>socket</i> argument does not refer to a socket.
39811	[EOPNOTSUPP]	The <i>socket</i> argument is associated with a socket that does not support one or
39812		more of the values set in <i>flags</i> .
39813	[EPIPE]	The socket is shut down for writing, or the socket is connection-mode and is
39814		no longer connected. In the latter case, and if the socket is of type
39815		SOCK_STREAM, the SIGPIPE signal is generated to the calling thread.
39816	If the address family of the socket is AF_UNIX, then <i>sendto()</i> shall fail if:	
39817	[EIO]	An I/O error occurred while reading from or writing to the file system.
39818	[ELOOP]	A loop exists in symbolic links encountered during resolution of the pathname
39819		in the socket address.
39820	[ENAMETOOLONG]	
39821		A component of a pathname exceeded {NAME_MAX} characters, or an entire
39822		pathname exceeded {PATH_MAX} characters.
39823	[ENOENT]	A component of the pathname does not name an existing file or the pathname
39824		is an empty string.
39825	[ENOTDIR]	A component of the path prefix of the pathname in the socket address is not a
39826		directory.
39827	The <i>sendto()</i> function may fail if:	
39828	[EACCES]	Search permission is denied for a component of the path prefix; or write
39829		access to the named socket is denied.
39830	[EDESTADDRREQ]	
39831		The socket is not connection-mode and does not have its peer address set, and
39832		no destination address was specified.
39833	[EHOSTUNREACH]	
39834		The destination host cannot be reached (probably because the host is down or
39835		a remote router cannot reach it).
39836	[EINVAL]	The <i>dest_len</i> argument is not a valid length for the address family.
39837	[EIO]	An I/O error occurred while reading from or writing to the file system.

39838	[EISCONN]	A destination address was specified and the socket is already connected. This error may or may not be returned for connection mode sockets.
39839		
39840	[ENETDOWN]	The local network interface used to reach the destination is down.
39841	[ENETUNREACH]	
39842		No route to the network is present.
39843	[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
39844	[ENOMEM]	Insufficient memory was available to fulfill the request.
39845	If the address family of the socket is AF_UNIX, then <i>sendto()</i> may fail if:	
39846	[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the pathname in the socket address.
39847		
39848	[ENAMETOOLONG]	
39849		Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.
39850		
39851	EXAMPLES	
39852	None.	
39853	APPLICATION USAGE	
39854	The <i>select()</i> and <i>poll()</i> functions can be used to determine when it is possible to send more data.	
39855	RATIONALE	
39856	None.	
39857	FUTURE DIRECTIONS	
39858	None.	
39859	SEE ALSO	
39860	<i>getsockopt()</i> , <i>poll()</i> , <i>recv()</i> , <i>recvfrom()</i> , <i>recvmsg()</i> , <i>select()</i> , <i>send()</i> , <i>sendmsg()</i> , <i>setsockopt()</i> ,	
39861	<i>shutdown()</i> , <i>socket()</i> , the Base Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>	
39862	CHANGE HISTORY	
39863	First released in Issue 6. Derived from the XNS, Issue 5.2 specification.	
39864	The wording of the mandatory [ELOOP] error condition is updated, and a second optional	
39865	[ELOOP] error condition is added.	

39866 NAME

39867 setbuf — assign buffering to a stream

39868 SYNOPSIS

39869 #include <stdio.h>

39870 void setbuf(FILE *restrict stream, char *restrict buf);

39871 DESCRIPTION

39872 cx The functionality described on this reference page is aligned with the ISO C standard. Any
39873 conflict between the requirements described here and the ISO C standard is unintentional. This
39874 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

39875 Except that it returns no value, the function call:

39876 setbuf(stream, buf)

39877 shall be equivalent to:

39878 setvbuf(stream, buf, _IOFBF, BUFSIZ)

39879 if *buf* is not a null pointer, or to:

39880 setvbuf(stream, buf, _IONBF, BUFSIZ)

39881 if *buf* is a null pointer.

39882 RETURN VALUE

39883 The *setbuf()* function shall not return a value.

39884 ERRORS

39885 No errors are defined.

39886 EXAMPLES

39887 None.

39888 APPLICATION USAGE

39889 A common source of error is allocating buffer space as an “automatic” variable in a code block,
39890 and then failing to close the stream in the same block.

39891 With *setbuf()*, allocating a buffer of BUFSIZ bytes does not necessarily imply that all of BUFSIZ
39892 bytes are used for the buffer area.

39893 RATIONALE

39894 None.

39895 FUTURE DIRECTIONS

39896 None.

39897 SEE ALSO

39898 *fopen()*, *setvbuf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

39899 CHANGE HISTORY

39900 First released in Issue 1. Derived from Issue 1 of the SVID.

39901 Issue 6

39902 The prototype for *setbuf()* is updated for alignment with the ISO/IEC 9899:1999 standard.

39903 NAME

39904 setcontext — set current user context

39905 SYNOPSIS

39906 xSI #include <ucontext.h>

39907 int setcontext(const ucontext_t *ucp);

39908

39909 DESCRIPTION

39910 Refer to *getcontext()*.

39911 **NAME**

39912 setegid — set the effective group ID

39913 **SYNOPSIS**

39914 #include <unistd.h>

39915 int setegid(gid_t *gid*);39916 **DESCRIPTION**

39917 If *gid* is equal to the real group ID or the saved set-group-ID, or if the process has appropriate
 39918 privileges, *setegid()* shall set the effective group ID of the calling process to *gid*; the real group
 39919 ID, saved set-group-ID, and any supplementary group IDs shall remain unchanged.

39920 The *setegid()* function shall not affect the supplementary group list in any way.

39921 **RETURN VALUE**

39922 Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned and *errno* set to
 39923 indicate the error.

39924 **ERRORS**

39925 The *setegid()* function shall fail if:

39926 [EINVAL] The value of the *gid* argument is invalid and is not supported by the
 39927 implementation.

39928 [EPERM] The process does not have appropriate privileges and *gid* does not match the
 39929 real group ID or the saved set-group-ID.

39930 **EXAMPLES**

39931 None.

39932 **APPLICATION USAGE**

39933 None.

39934 **RATIONALE**39935 Refer to the RATIONALE section in *setuid()*.39936 **FUTURE DIRECTIONS**

39937 None.

39938 **SEE ALSO**

39939 *exec*, *getegid()*, *geteuid()*, *getgid()*, *getuid()*, *seteuid()*, *setgid()*, *setregid()*, *setreuid()*, *setuid()*, the
 39940 Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

39941 **CHANGE HISTORY**

39942 First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

39943 **NAME**

39944 setenv — add or change environment variable

39945 **SYNOPSIS**39946 **CX** #include <stdlib.h>

39947 int setenv(const char *envname, const char *envval, int overwrite);

39948

39949 **DESCRIPTION**

39950 The *setenv()* function shall update or add a variable in the environment of the calling process.
 39951 The *envname* argument points to a string containing the name of an environment variable to be
 39952 added or altered. The environment variable shall be set to the value to which *envval* points. The
 39953 function shall fail if *envname* points to a string which contains an '=' character. If the
 39954 environment variable named by *envname* already exists and the value of *overwrite* is non-zero,
 39955 the function shall return success and the environment shall be updated. If the environment
 39956 variable named by *envname* already exists and the value of *overwrite* is zero, the function shall
 39957 return success and the environment shall remain unchanged.

39958 If the application modifies *environ* or the pointers to which it points, the behavior of *setenv()* is
 39959 undefined. The *setenv()* function shall update the list of pointers to which *environ* points.

39960 The strings described by *envname* and *envval* are copied by this function.

39961 The *setenv()* function need not be reentrant. A function that is not required to be reentrant is not
 39962 required to be thread-safe.

39963 **RETURN VALUE**

39964 Upon successful completion, zero shall be returned. Otherwise, -1 shall be returned, *errno* set to
 39965 indicate the error, and the environment shall be unchanged.

39966 **ERRORS**

39967 The *setenv()* function shall fail if:

39968 [EINVAL] The *name* argument is a null pointer, points to an empty string, or points to a
 39969 string containing an '=' character.

39970 [ENOMEM] Insufficient memory was available to add a variable or its value to the
 39971 environment.

39972 **EXAMPLES**

39973 None.

39974 **APPLICATION USAGE**

39975 None.

39976 **RATIONALE**

39977 Unanticipated results may occur if *setenv()* changes the external variable *environ*. In particular,
 39978 if the optional *envp* argument to *main()* is present, it is not changed, and thus may point to an
 39979 obsolete copy of the environment (as may any other copy of *environ*). However, other than the
 39980 aforementioned restriction, the developers of IEEE Std 1003.1-2001 intended that the traditional
 39981 method of walking through the environment by way of the *environ* pointer must be supported.

39982 It was decided that *setenv()* should be required by this revision because it addresses a piece of
 39983 missing functionality, and does not impose a significant burden on the implementor.

39984 There was considerable debate as to whether the System V *putenv()* function or the BSD *setenv()*
 39985 function should be required as a mandatory function. The *setenv()* function was chosen because
 39986 it permitted the implementation of the *unsetenv()* function to delete environmental variables,
 39987 without specifying an additional interface. The *putenv()* function is available as an XSI

39988 extension.

39989 The standard developers considered requiring that *setenv()* indicate an error when a call to it
39990 would result in exceeding {ARG_MAX}. The requirement was rejected since the condition might
39991 be temporary, with the application eventually reducing the environment size. The ultimate
39992 success or failure depends on the size at the time of a call to *exec*, which returns an indication of
39993 this error condition.

39994 **FUTURE DIRECTIONS**

39995 None.

39996 **SEE ALSO**

39997 *getenv()*, *unsetenv()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<stdlib.h>**,
39998 **<sys/types.h>**, **<unistd.h>**

39999 **CHANGE HISTORY**

40000 First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

40001 **NAME**

40002 seteuid — set effective user ID

40003 **SYNOPSIS**

40004 #include <unistd.h>

40005 int seteuid(uid_t uid);

40006 **DESCRIPTION**

40007 If *uid* is equal to the real user ID or the saved set-user-ID, or if the process has appropriate privileges, *seteuid()* shall set the effective user ID of the calling process to *uid*; the real user ID and saved set-user-ID shall remain unchanged.

40010 The *seteuid()* function shall not affect the supplementary group list in any way.

40011 **RETURN VALUE**

40012 Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned and *errno* set to indicate the error.

40014 **ERRORS**40015 The *seteuid()* function shall fail if:

40016 [EINVAL] The value of the *uid* argument is invalid and is not supported by the implementation.

40018 [EPERM] The process does not have appropriate privileges and *uid* does not match the real group ID or the saved set-group-ID.

40020 **EXAMPLES**

40021 None.

40022 **APPLICATION USAGE**

40023 None.

40024 **RATIONALE**40025 Refer to the RATIONALE section in *setuid()*.40026 **FUTURE DIRECTIONS**

40027 None.

40028 **SEE ALSO**

40029 *exec*, *getegid()*, *geteuid()*, *getgid()*, *getuid()*, *setegid()*, *setgid()*, *setregid()*, *setreuid()*, *setuid()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

40031 **CHANGE HISTORY**

40032 First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

40033 **NAME**

40034 setgid — set-group-ID

40035 **SYNOPSIS**

40036 #include <unistd.h>

40037 int setgid(gid_t gid);

40038 **DESCRIPTION**40039 If the process has appropriate privileges, *setgid()* shall set the real group ID, effective group ID,
40040 and the saved set-group-ID of the calling process to *gid*.40041 If the process does not have appropriate privileges, but *gid* is equal to the real group ID or the
40042 saved set-group-ID, *setgid()* shall set the effective group ID to *gid*; the real group ID and saved
40043 set-group-ID shall remain unchanged.40044 The *setgid()* function shall not affect the supplementary group list in any way.

40045 Any supplementary group IDs of the calling process shall remain unchanged.

40046 **RETURN VALUE**40047 Upon successful completion, 0 is returned. Otherwise, -1 shall be returned and *errno* set to
40048 indicate the error.40049 **ERRORS**40050 The *setgid()* function shall fail if:40051 [EINVAL] The value of the *gid* argument is invalid and is not supported by the
40052 implementation.40053 [EPERM] The process does not have appropriate privileges and *gid* does not match the
40054 real group ID or the saved set-group-ID.40055 **EXAMPLES**

40056 None.

40057 **APPLICATION USAGE**

40058 None.

40059 **RATIONALE**40060 Refer to the RATIONALE section in *setuid()*.40061 **FUTURE DIRECTIONS**

40062 None.

40063 **SEE ALSO**40064 *exec*, *getegid()*, *geteuid()*, *getgid()*, *getuid()*, *setegid()*, *seteuid()*, *setregid()*, *setreuid()*, *setuid()*, the
40065 Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>40066 **CHANGE HISTORY**

40067 First released in Issue 1. Derived from Issue 1 of the SVID.

40068 **Issue 6**

40069 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

40070 The following new requirements on POSIX implementations derive from alignment with the
40071 Single UNIX Specification:

- 40072
- The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
40073 required for conforming implementations of previous POSIX specifications, it was not
40074 required for UNIX applications.

- 40075 • Functionality associated with `_POSIX_SAVED_IDS` is now mandated. This is a FIPS
- 40076 requirement.
- 40077 The following changes were made to align with the IEEE P1003.1a draft standard:
- 40078 • The effects of *setgid()* in processes without appropriate privileges are changed.
- 40079 • A requirement that the supplementary group list is not affected is added.

40080 **NAME**

40081 setgrent — reset the group database to the first entry

40082 **SYNOPSIS**

40083 xSI #include <grp.h>

40084 void setgrent(void);

40085

40086 **DESCRIPTION**

40087 Refer to *endgrent()*.

40088 NAME

40089 sethostent — network host database functions

40090 SYNOPSIS

40091 #include <netdb.h>

40092 void sethostent(int *stayopen*);

40093 DESCRIPTION

40094 Refer to *endhostent()*.

40095 **NAME**

40096 setitimer — set the value of an interval timer

40097 **SYNOPSIS**40098 XSI `#include <sys/time.h>`40099 `int setitimer(int which, const struct itimerval *restrict value,`
40100 `struct itimerval *restrict ovalue);`

40101

40102 **DESCRIPTION**40103 Refer to *getitimer()*.

40104 **NAME**

40105 setjmp — set jump point for a non-local goto

40106 **SYNOPSIS**

40107 #include <setjmp.h>

40108 int setjmp(jmp_buf env);

40109 **DESCRIPTION**

40110 cx The functionality described on this reference page is aligned with the ISO C standard. Any
 40111 conflict between the requirements described here and the ISO C standard is unintentional. This
 40112 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

40113 A call to *setjmp()* shall save the calling environment in its *env* argument for later use by
 40114 *longjmp()*.

40115 It is unspecified whether *setjmp()* is a macro or a function. If a macro definition is suppressed in
 40116 order to access an actual function, or a program defines an external identifier with the name
 40117 *setjmp*, the behavior is undefined.

40118 An application shall ensure that an invocation of *setjmp()* appears in one of the following
 40119 contexts only:

- 40120 • The entire controlling expression of a selection or iteration statement
- 40121 • One operand of a relational or equality operator with the other operand an integral constant
 40122 expression, with the resulting expression being the entire controlling expression of a
 40123 selection or iteration statement
- 40124 • The operand of a unary '!' operator with the resulting expression being the entire
 40125 controlling expression of a selection or iteration
- 40126 • The entire expression of an expression statement (possibly cast to **void**)

40127 If the invocation appears in any other context, the behavior is undefined.

40128 **RETURN VALUE**

40129 If the return is from a direct invocation, *setjmp()* shall return 0. If the return is from a call to
 40130 *longjmp()*, *setjmp()* shall return a non-zero value.

40131 **ERRORS**

40132 No errors are defined.

40133 **EXAMPLES**

40134 None.

40135 **APPLICATION USAGE**

40136 In general, *sigsetjmp()* is more useful in dealing with errors and interrupts encountered in a low-
 40137 level subroutine of a program.

40138 **RATIONALE**

40139 None.

40140 **FUTURE DIRECTIONS**

40141 None.

40142 **SEE ALSO**

40143 *longjmp()*, *sigsetjmp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <setjmp.h>

40144 CHANGE HISTORY

40145 First released in Issue 1. Derived from Issue 1 of the SVID.

40146 Issue 6

40147 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

40148 **NAME**40149 setkey — set encoding key (**CRYPT**)40150 **SYNOPSIS**

40151 xSI #include <stdlib.h>

40152 void setkey(const char *key);

40153

40154 **DESCRIPTION**

40155 The *setkey()* function provides access to an implementation-defined encoding algorithm. The
40156 argument of *setkey()* is an array of length 64 bytes containing only the bytes with numerical
40157 value of 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is
40158 ignored; this gives a 56-bit key which is used by the algorithm. This is the key that shall be used
40159 with the algorithm to encode a string *block* passed to *encrypt()*.

40160 The *setkey()* function shall not change the setting of *errno* if successful. An application wishing to
40161 check for error situations should set *errno* to 0 before calling *setkey()*. If *errno* is non-zero on
40162 return, an error has occurred.

40163 The *setkey()* function need not be reentrant. A function that is not required to be reentrant is not
40164 required to be thread-safe.

40165 **RETURN VALUE**

40166 No values are returned.

40167 **ERRORS**40168 The *setkey()* function shall fail if:

40169 [ENOSYS] The functionality is not supported on this implementation.

40170 **EXAMPLES**

40171 None.

40172 **APPLICATION USAGE**

40173 Decoding need not be implemented in all environments. This is related to government
40174 restrictions in some countries on encryption and decryption routines. Historical practice has
40175 been to ship a different version of the encryption library without the decryption feature in the
40176 routines supplied. Thus the exported version of *encrypt()* does encoding but not decoding.

40177 **RATIONALE**

40178 None.

40179 **FUTURE DIRECTIONS**

40180 None.

40181 **SEE ALSO**40182 *crypt()*, *encrypt()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>40183 **CHANGE HISTORY**

40184 First released in Issue 1. Derived from Issue 1 of the SVID.

40185 **Issue 5**40186 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

40187 **NAME**

40188 setlocale — set program locale

40189 **SYNOPSIS**

40190 #include <locale.h>

40191 char *setlocale(int *category*, const char **locale*);40192 **DESCRIPTION**

40193 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 40194 conflict between the requirements described here and the ISO C standard is unintentional. This
 40195 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

40196 The *setlocale()* function selects the appropriate piece of the program's locale, as specified by the
 40197 *category* and *locale* arguments, and may be used to change or query the program's entire locale or
 40198 portions thereof. The value *LC_ALL* for *category* names the program's entire locale; other values
 40199 for *category* name only a part of the program's locale:

40200 *LC_COLLATE* Affects the behavior of regular expressions and the collation functions.

40201 *LC_CTYPE* Affects the behavior of regular expressions, character classification, character
 40202 conversion functions, and wide-character functions.

40203 CX *LC_MESSAGES* Affects what strings are expected by commands and utilities as affirmative or
 40204 negative responses.

40205 XSI It also affects what strings are given by commands and utilities as affirmative
 40206 or negative responses, and the content of messages.

40207 *LC_MONETARY* Affects the behavior of functions that handle monetary values.

40208 *LC_NUMERIC* Affects the behavior of functions that handle numeric values.

40209 *LC_TIME* Affects the behavior of the time conversion functions.

40210 The *locale* argument is a pointer to a character string containing the required setting of *category*.
 40211 The contents of this string are implementation-defined. In addition, the following preset values
 40212 of *locale* are defined for all settings of *category*:

40213 CX "POSIX" Specifies the minimal environment for C-language translation called the
 40214 POSIX locale. If *setlocale()* is not invoked, the POSIX locale is the default at
 40215 entry to *main()*.

40216 "C" Equivalent to "POSIX".

40217 CX "" Specifies an implementation-defined native environment. This corresponds to
 40218 the value of the associated environment variables, *LC_** and *LANG*; see the
 40219 Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale and the
 40220 Base Definitions volume of IEEE Std 1003.1-2001, Chapter 8, Environment
 40221 Variables.

40222 A null pointer Used to direct *setlocale()* to query the current internationalized environment
 40223 and return the name of the *locale()*.

40224 THR The locale state is common to all threads within a process.

40225 **RETURN VALUE**

40226 Upon successful completion, *setlocale()* shall return the string associated with the specified
 40227 category for the new locale. Otherwise, *setlocale()* shall return a null pointer and the program's
 40228 locale is not changed.

40229 A null pointer for *locale* causes *setlocale()* to return a pointer to the string associated with the
 40230 *category* for the program's current locale. The program's locale shall not be changed.

40231 The string returned by *setlocale()* is such that a subsequent call with that string and its associated
 40232 *category* shall restore that part of the program's locale. The application shall not modify the string
 40233 returned which may be overwritten by a subsequent call to *setlocale()*.

40234 ERRORS

40235 No errors are defined.

40236 EXAMPLES

40237 None.

40238 APPLICATION USAGE

40239 The following code illustrates how a program can initialize the international environment for
 40240 one language, while selectively modifying the program's locale such that regular expressions
 40241 and string operations can be applied to text recorded in a different language:

```
40242 setlocale(LC_ALL, "De");
40243 setlocale(LC_COLLATE, "Fr@dict");
```

40244 Internationalized programs must call *setlocale()* to initiate a specific language operation. This can
 40245 be done by calling *setlocale()* as follows:

```
40246 setlocale(LC_ALL, "");
```

40247 Changing the setting of *LC_MESSAGES* has no effect on catalogs that have already been opened
 40248 by calls to *catopen()*.

40249 RATIONALE

40250 The ISO C standard defines a collection of functions to support internationalization. One of the
 40251 most significant aspects of these functions is a facility to set and query the *international*
 40252 *environment*. The international environment is a repository of information that affects the
 40253 behavior of certain functionality, namely:

- 40254 1. Character handling
- 40255 2. Collating
- 40256 3. Date/time formatting
- 40257 4. Numeric editing
- 40258 5. Monetary formatting
- 40259 6. Messaging

40260 The *setlocale()* function provides the application developer with the ability to set all or portions,
 40261 called *categories*, of the international environment. These categories correspond to the areas of
 40262 functionality mentioned above. The syntax for *setlocale()* is as follows:

```
40263 char *setlocale(int category, const char *locale);
```

40264 where *category* is the name of one of following categories, namely:

```
40265 LC_COLLATE
40266 LC_CTYPE
40267 LC_MESSAGES
40268 LC_MONETARY
40269 LC_NUMERIC
40270 LC_TIME
```


In addition, a special value called *LC_ALL* directs *setlocale()* to set all categories.

There are two primary uses of *setlocale()*:

1. Querying the international environment to find out what it is set to
2. Setting the international environment, or *locale*, to a specific value

The behavior of *setlocale()* in these two areas is described below. Since it is difficult to describe the behavior in words, examples are used to illustrate the behavior of specific uses.

To query the international environment, *setlocale()* is invoked with a specific category and the NULL pointer as the locale. The NULL pointer is a special directive to *setlocale()* that tells it to query rather than set the international environment. The following syntax is used to query the name of the international environment:

```
setlocale({LC_ALL, LC_COLLATE, LC_CTYPE, LC_MESSAGES, LC_MONETARY, \
          LC_NUMERIC, LC_TIME}, (char *) NULL);
```

The *setlocale()* function shall return the string corresponding to the current international environment. This value may be used by a subsequent call to *setlocale()* to reset the international environment to this value. However, it should be noted that the return value from *setlocale()* may be a pointer to a static area within the function and is not guaranteed to remain unchanged (that is, it may be modified by a subsequent call to *setlocale()*). Therefore, if the purpose of calling *setlocale()* is to save the value of the current international environment so it can be changed and reset later, the return value should be copied to an array of **char** in the calling program.

There are three ways to set the international environment with *setlocale()*:

setlocale(category, string)

This usage sets a specific *category* in the international environment to a specific value corresponding to the value of the *string*. A specific example is provided below:

```
setlocale(LC_ALL, "fr_FR.ISO-8859-1");
```

In this example, all categories of the international environment are set to the locale corresponding to the string "fr_FR.ISO-8859-1", or to the French language as spoken in France using the ISO/IEC 8859-1: 1998 standard codeset.

If the string does not correspond to a valid locale, *setlocale()* shall return a NULL pointer and the international environment is not changed. Otherwise, *setlocale()* shall return the name of the locale just set.

setlocale(category, "C")

The ISO C standard states that one locale must exist on all conforming implementations. The name of the locale is C and corresponds to a minimal international environment needed to support the C programming language.

setlocale(category, "")

This sets a specific category to an implementation-defined default. This corresponds to the value of the environment variables.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *isalnum()*, *isalpha()*, *isblank()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*, *iswalnum()*, *iswalpha()*, *iswblank()*, *iswcntrl()*, *iswctype()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*, *iswpunct()*, *iswspace()*, *iswupper()*, *iswxdigit()*, *isxdigit()*,

40315 *localeconv()*, *mblen()*, *mbstowcs()*, *mbtowc()*, *nl_langinfo()*, *printf()*, *scanf()*, *setlocale()*, *strcoll()*,
40316 *strerror()*, *strfmon()*, *strtod()*, *strxfrm()*, *tolower()*, *toupper()*, *towlower()*, *towupper()*, *wscoll()*,
40317 *wctod()*, *wcstombs()*, *wcsxfrm()*, *wctomb()*, the Base Definitions volume of IEEE Std 1003.1-2001,
40318 **<langinfo.h>**, **<locale.h>**

40319 **CHANGE HISTORY**

40320 First released in Issue 3.

40321 **Issue 5**

40322 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

40323 **Issue 6**

40324 Extensions beyond the ISO C standard are marked.

40325 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

40326 **NAME**

40327 setlogmask — set the log priority mask

40328 **SYNOPSIS**

40329 xSI #include <syslog.h>

40330 int setlogmask(int *maskpri*);

40331

40332 **DESCRIPTION**40333 Refer to *closelog()*.

40334 NAME

40335 setnetent — network database function

40336 SYNOPSIS

40337 #include <netdb.h>

40338 void setnetent(int stayopen);

40339 DESCRIPTION

40340 Refer to *endnetent()*.

40341 **NAME**

40342 setpgid — set process group ID for job control

40343 **SYNOPSIS**

40344 #include <unistd.h>

40345 int setpgid(pid_t pid, pid_t pgid);

40346 **DESCRIPTION**

40347 The *setpgid()* function shall either join an existing process group or create a new process group
 40348 within the session of the calling process. The process group ID of a session leader shall not
 40349 change. Upon successful completion, the process group ID of the process with a process ID that
 40350 matches *pid* shall be set to *pgid*. As a special case, if *pid* is 0, the process ID of the calling process
 40351 shall be used. Also, if *pgid* is 0, the process group ID of the indicated process shall be used.

40352 **RETURN VALUE**

40353 Upon successful completion, *setpgid()* shall return 0; otherwise, -1 shall be returned and *errno*
 40354 shall be set to indicate the error.

40355 **ERRORS**40356 The *setpgid()* function shall fail if:

40357 [EACCES] The value of the *pid* argument matches the process ID of a child process of the
 40358 calling process and the child process has successfully executed one of the *exec*
 40359 functions.

40360 [EINVAL] The value of the *pgid* argument is less than 0, or is not a value supported by
 40361 the implementation.

40362 [EPERM] The process indicated by the *pid* argument is a session leader.

40363 [EPERM] The value of the *pid* argument matches the process ID of a child process of the
 40364 calling process and the child process is not in the same session as the calling
 40365 process.

40366 [EPERM] The value of the *pgid* argument is valid but does not match the process ID of
 40367 the process indicated by the *pid* argument and there is no process with a
 40368 process group ID that matches the value of the *pgid* argument in the same
 40369 session as the calling process.

40370 [ESRCH] The value of the *pid* argument does not match the process ID of the calling
 40371 process or of a child process of the calling process.

40372 **EXAMPLES**

40373 None.

40374 **APPLICATION USAGE**

40375 None.

40376 **RATIONALE**

40377 The *setpgid()* function shall group processes together for the purpose of signaling, placement in
 40378 foreground or background, and other job control actions.

40379 The *setpgid()* function is similar to the *setpgrp()* function of 4.2 BSD, except that 4.2 BSD allowed
 40380 the specified new process group to assume any value. This presents certain security problems
 40381 and is more flexible than necessary to support job control.

40382 To provide tighter security, *setpgid()* only allows the calling process to join a process group
 40383 already in use inside its session or create a new process group whose process group ID was
 40384 equal to its process ID.

40385 When a job control shell spawns a new job, the processes in the job must be placed into a new
 40386 process group via *setpgid()*. There are two timing constraints involved in this action:

- 40387 1. The new process must be placed in the new process group before the appropriate program
 40388 is launched via one of the *exec* functions.
- 40389 2. The new process must be placed in the new process group before the shell can correctly
 40390 send signals to the new process group.

40391 To address these constraints, the following actions are performed. The new processes call
 40392 *setpgid()* to alter their own process groups after *fork()* but before *exec*. This satisfies the first
 40393 constraint. Under 4.3 BSD, the second constraint is satisfied by the synchronization property of
 40394 *vfork()*; that is, the shell is suspended until the child has completed the *exec*, thus ensuring that
 40395 the child has completed the *setpgid()*. A new version of *fork()* with this same synchronization
 40396 property was considered, but it was decided instead to merely allow the parent shell process to
 40397 adjust the process group of its child processes via *setpgid()*. Both timing constraints are now
 40398 satisfied by having both the parent shell and the child attempt to adjust the process group of the
 40399 child process; it does not matter which succeeds first.

40400 Since it would be confusing to an application to have its process group change after it began
 40401 executing (that is, after *exec*), and because the child process would already have adjusted its
 40402 process group before this, the [EACCES] error was added to disallow this.

40403 One non-obvious use of *setpgid()* is to allow a job control shell to return itself to its original
 40404 process group (the one in effect when the job control shell was executed). A job control shell
 40405 does this before returning control back to its parent when it is terminating or suspending itself as
 40406 a way of restoring its job control “state” back to what its parent would expect. (Note that the
 40407 original process group of the job control shell typically matches the process group of its parent,
 40408 but this is not necessarily always the case.)

40409 FUTURE DIRECTIONS

40410 None.

40411 SEE ALSO

40412 *exec*, *getpgrp()*, *setsid()*, *tcsetpgrp()*, the Base Definitions volume of IEEE Std 1003.1-2001,
 40413 *<sys/types.h>*, *<unistd.h>*

40414 CHANGE HISTORY

40415 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

40416 Issue 6

40417 In the SYNOPSIS, the optional include of the *<sys/types.h>* header is removed.

40418 The following new requirements on POSIX implementations derive from alignment with the
 40419 Single UNIX Specification:

- 40420 • The requirement to include *<sys/types.h>* has been removed. Although *<sys/types.h>* was
 40421 required for conforming implementations of previous POSIX specifications, it was not
 40422 required for UNIX applications.
- 40423 • The *setpgid()* function is mandatory since *_POSIX_JOB_CONTROL* is required to be defined
 40424 in this issue. This is a FIPS requirement.

40425 **NAME**

40426 setpgrp — set the process group ID

40427 **SYNOPSIS**

40428 XSI #include <unistd.h>

40429 pid_t setpgrp(void);

40430

40431 **DESCRIPTION**

40432 If the calling process is not already a session leader, *setpgrp()* sets the process group ID of the
40433 calling process to the process ID of the calling process. If *setpgrp()* creates a new session, then
40434 the new session has no controlling terminal.

40435 The *setpgrp()* function has no effect when the calling process is a session leader.

40436 **RETURN VALUE**40437 Upon completion, *setpgrp()* shall return the process group ID.40438 **ERRORS**

40439 No errors are defined.

40440 **EXAMPLES**

40441 None.

40442 **APPLICATION USAGE**

40443 None.

40444 **RATIONALE**

40445 None.

40446 **FUTURE DIRECTIONS**

40447 None.

40448 **SEE ALSO**

40449 *exec*, *fork()*, *getpid()*, *getsid()*, *kill()*, *setpgid()*, *setsid()*, the Base Definitions volume of
40450 IEEE Std 1003.1-2001, <unistd.h>

40451 **CHANGE HISTORY**

40452 First released in Issue 4, Version 2.

40453 **Issue 5**

40454 Moved from X/OPEN UNIX extension to BASE.

40455 NAME

40456 setpriority — set the nice value

40457 SYNOPSIS

40458 xSI #include <sys/resource.h>

40459 int setpriority(int *which*, id_t *who*, int *nice*);

40460

40461 DESCRIPTION

40462 Refer to *getpriority()*.

40463 **NAME**

40464 setprotoent — network protocol database functions

40465 **SYNOPSIS**

40466 #include <netdb.h>

40467 void setprotoent(int *stayopen*);40468 **DESCRIPTION**40469 Refer to *endprotoent()*.

40470 NAME

40471 setpwent — user database function

40472 SYNOPSIS

40473 XSI #include <pwd.h>

40474 void setpwent(void);

40475

40476 DESCRIPTION

40477 Refer to *endpwent()*.

40478 **NAME**

40479 setregid — set real and effective group IDs

40480 **SYNOPSIS**

40481 XSI #include <unistd.h>

40482 int setregid(gid_t rgid, gid_t egid);

40483

40484 **DESCRIPTION**40485 The *setregid()* function shall set the real and effective group IDs of the calling process.40486 If *rgid* is *-1*, the real group ID shall not be changed; if *egid* is *-1*, the effective group ID shall not
40487 be changed.

40488 The real and effective group IDs may be set to different values in the same call.

40489 Only a process with appropriate privileges can set the real group ID and the effective group ID
40490 to any valid value.40491 A non-privileged process can set either the real group ID to the saved set-group-ID from one of
40492 the *exec* family of functions, or the effective group ID to the saved set-group-ID or the real group
40493 ID.

40494 Any supplementary group IDs of the calling process remain unchanged.

40495 **RETURN VALUE**40496 Upon successful completion, 0 shall be returned. Otherwise, *-1* shall be returned and *errno* set to
40497 indicate the error, and neither of the group IDs are changed.40498 **ERRORS**40499 The *setregid()* function shall fail if:40500 [EINVAL] The value of the *rgid* or *egid* argument is invalid or out-of-range.40501 [EPERM] The process does not have appropriate privileges and a change other than
40502 changing the real group ID to the saved set-group-ID, or changing the
40503 effective group ID to the real group ID or the saved set-group-ID, was
40504 requested.40505 **EXAMPLES**

40506 None.

40507 **APPLICATION USAGE**40508 If a set-group-ID process sets its effective group ID to its real group ID, it can still set its effective
40509 group ID back to the saved set-group-ID.40510 **RATIONALE**

40511 None.

40512 **FUTURE DIRECTIONS**

40513 None.

40514 **SEE ALSO**40515 *exec*, *getegid()*, *geteuid()*, *getgid()*, *getuid()*, *setegid()*, *seteuid()*, *setgid()*, *setreuid()*, *setuid()*, the
40516 Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>40517 **CHANGE HISTORY**

40518 First released in Issue 4, Version 2.

40519 Issue 5

40520 Moved from X/OPEN UNIX extension to BASE.

40521 The DESCRIPTION is updated to indicate that the saved set-group-ID can be set by any of the
40522 *exec* family of functions, not just *execve()*.

40523 **NAME**

40524 setreuid — set real and effective user IDs

40525 **SYNOPSIS**

40526 XSI #include <unistd.h>

40527 int setreuid(uid_t ruid, uid_t euid);

40528

40529 **DESCRIPTION**

40530 The *setreuid()* function shall set the real and effective user IDs of the current process to the
 40531 values specified by the *ruid* and *euid* arguments. If *ruid* or *euid* is *-1*, the corresponding effective
 40532 or real user ID of the current process shall be left unchanged.

40533 A process with appropriate privileges can set either ID to any value. An unprivileged process
 40534 can only set the effective user ID if the *euid* argument is equal to either the real, effective, or
 40535 saved user ID of the process.

40536 It is unspecified whether a process without appropriate privileges is permitted to change the real
 40537 user ID to match the current real, effective, or saved set-user-ID of the process.

40538 **RETURN VALUE**

40539 Upon successful completion, 0 shall be returned. Otherwise, *-1* shall be returned and *errno* set to
 40540 indicate the error.

40541 **ERRORS**40542 The *setreuid()* function shall fail if:

40543 [EINVAL] The value of the *ruid* or *euid* argument is invalid or out-of-range.

40544 [EPERM] The current process does not have appropriate privileges, and either an
 40545 attempt was made to change the effective user ID to a value other than the
 40546 real user ID or the saved set-user-ID or an attempt was made to change the
 40547 real user ID to a value not permitted by the implementation.

40548 **EXAMPLES**40549 **Setting the Effective User ID to the Real User ID**

40550 The following example sets the effective user ID of the calling process to the real user ID, so that
 40551 files created later will be owned by the current user.

40552 #include <unistd.h>

40553 #include <sys/types.h>

40554 ...

40555 setreuid(getuid(), getuid());

40556 ...

40557 **APPLICATION USAGE**

40558 None.

40559 **RATIONALE**

40560 None.

40561 **FUTURE DIRECTIONS**

40562 None.

40563 **SEE ALSO**

40564 *getegid()*, *geteuid()*, *getgid()*, *getuid()*, *setegid()*, *seteuid()*, *setgid()*, *setregid()*, *setuid()*, the Base
40565 Definitions volume of IEEE Std 1003.1-2001, <**unistd.h**>

40566 **CHANGE HISTORY**

40567 First released in Issue 4, Version 2.

40568 **Issue 5**

40569 Moved from X/OPEN UNIX extension to BASE.

40570 **NAME**

40571 setrlimit — control maximum resource consumption

40572 **SYNOPSIS**

40573 XSI #include <sys/resource.h>

40574 int setrlimit(int resource, const struct rlimit *rlp);

40575

40576 **DESCRIPTION**40577 Refer to *getrlimit()*.

40578 NAME

40579 setservent — network services database functions

40580 SYNOPSIS

40581 #include <netdb.h>

40582 void setservent(int *stayopen*);

40583 DESCRIPTION

40584 Refer to *endservent()*.

40585 **NAME**

40586 setsid — create session and set process group ID

40587 **SYNOPSIS**

40588 #include <unistd.h>

40589 pid_t setsid(void);

40590 **DESCRIPTION**

40591 The *setsid()* function shall create a new session, if the calling process is not a process group
40592 leader. Upon return the calling process shall be the session leader of this new session, shall be
40593 the process group leader of a new process group, and shall have no controlling terminal. The
40594 process group ID of the calling process shall be set equal to the process ID of the calling process.
40595 The calling process shall be the only process in the new process group and the only process in
40596 the new session.

40597 **RETURN VALUE**

40598 Upon successful completion, *setsid()* shall return the value of the new process group ID of the
40599 calling process. Otherwise, it shall return (**pid_t**)−1 and set *errno* to indicate the error.

40600 **ERRORS**40601 The *setsid()* function shall fail if:

40602 [EPERM]	The calling process is already a process group leader, or the process group ID
40603	of a process other than the calling process matches the process ID of the
40604	calling process.

40605 **EXAMPLES**

40606 None.

40607 **APPLICATION USAGE**

40608 None.

40609 **RATIONALE**

40610 The *setsid()* function is similar to the *setpgp()* function of System V. System V, without job
40611 control, groups processes into process groups and creates new process groups via *setpgp()*; only
40612 one process group may be part of a login session.

40613 Job control allows multiple process groups within a login session. In order to limit job control
40614 actions so that they can only affect processes in the same login session, this volume of
40615 IEEE Std 1003.1-2001 adds the concept of a session that is created via *setsid()*. The *setsid()*
40616 function also creates the initial process group contained in the session. Additional process
40617 groups can be created via the *setpgid()* function. A System V process group would correspond to
40618 a POSIX System Interfaces session containing a single POSIX process group. Note that this
40619 function requires that the calling process not be a process group leader. The usual way to ensure
40620 this is true is to create a new process with *fork()* and have it call *setsid()*. The *fork()* function
40621 guarantees that the process ID of the new process does not match any existing process group ID.

40622 **FUTURE DIRECTIONS**

40623 None.

40624 **SEE ALSO**

40625 *getsid()*, *setpgid()*, *setpgp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>,
40626 <unistd.h>

40627 **CHANGE HISTORY**

40628 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

40629 **Issue 6**

40630 In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

40631 The following new requirements on POSIX implementations derive from alignment with the
40632 Single UNIX Specification:

- 40633 • The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was
40634 required for conforming implementations of previous POSIX specifications, it was not
40635 required for UNIX applications.

40636 **NAME**

40637 setsockopt — set the socket options

40638 **SYNOPSIS**

40639 #include <sys/socket.h>

```
40640       int setsockopt(int socket, int level, int option_name,
40641                      const void *option_value, socklen_t option_len);
```

40642 **DESCRIPTION**

40643 The *setsockopt()* function shall set the option specified by the *option_name* argument, at the
 40644 protocol level specified by the *level* argument, to the value pointed to by the *option_value*
 40645 argument for the socket associated with the file descriptor specified by the *socket* argument.

40646 The *level* argument specifies the protocol level at which the option resides. To set options at the
 40647 socket level, specify the *level* argument as SOL_SOCKET. To set options at other levels, supply
 40648 the appropriate *level* identifier for the protocol controlling the option. For example, to indicate
 40649 that an option is interpreted by the TCP (Transport Control Protocol), set *level* to IPPROTO_TCP
 40650 as defined in the <netinet/in.h> header.

40651 The *option_name* argument specifies a single option to set. The *option_name* argument and any
 40652 specified options are passed uninterpreted to the appropriate protocol module for
 40653 interpretations. The <sys/socket.h> header defines the socket-level options. The options are as
 40654 follows:

40655 SO_DEBUG Turns on recording of debugging information. This option enables or
 40656 disables debugging in the underlying protocol modules. This option takes
 40657 an **int** value. This is a Boolean option.

40658 SO_BROADCAST Permits sending of broadcast messages, if this is supported by the
 40659 protocol. This option takes an **int** value. This is a Boolean option.

40660 SO_REUSEADDR Specifies that the rules used in validating addresses supplied to *bind()*
 40661 should allow reuse of local addresses, if this is supported by the protocol.
 40662 This option takes an **int** value. This is a Boolean option.

40663 SO_KEEPALIVE Keeps connections active by enabling the periodic transmission of
 40664 messages, if this is supported by the protocol. This option takes an **int**
 40665 value.

40666 If the connected socket fails to respond to these messages, the connection
 40667 is broken and threads writing to that socket are notified with a SIGPIPE
 40668 signal. This is a Boolean option.

40669 SO_LINGER Lingers on a *close()* if data is present. This option controls the action
 40670 taken when unsent messages queue on a socket and *close()* is performed.
 40671 If SO_LINGER is set, the system shall block the process during *close()*
 40672 until it can transmit the data or until the time expires. If SO_LINGER is
 40673 not specified, and *close()* is issued, the system handles the call in a way
 40674 that allows the process to continue as quickly as possible. This option
 40675 takes a **linger** structure, as defined in the <sys/socket.h> header, to
 40676 specify the state of the option and linger interval.

40677 SO_OOINLINE Leaves received out-of-band data (data marked urgent) inline. This
 40678 option takes an **int** value. This is a Boolean option.

40679 SO_SNDBUF Sets send buffer size. This option takes an **int** value.

40680	SO_RCVBUF	Sets receive buffer size. This option takes an int value.	
40681	SO_DONTROUTE	Requests that outgoing messages bypass the standard routing facilities. The destination shall be on a directly-connected network, and messages are directed to the appropriate network interface according to the destination address. The effect, if any, of this option depends on what protocol is in use. This option takes an int value. This is a Boolean option.	
40682			
40683			
40684			
40685			
40686	SO_RCVLOWAT	Sets the minimum number of bytes to process for socket input operations. The default value for SO_RCVLOWAT is 1. If SO_RCVLOWAT is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. (They may return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different from that returned; for example, out-of-band data.) This option takes an int value. Note that not all implementations allow this option to be set.	
40687			
40688			
40689			
40690			
40691			
40692			
40693			
40694	SO_RCVTIMEO	Sets the timeout value that specifies the maximum amount of time an input function waits until it completes. It accepts a timeval structure with the number of seconds and microseconds specifying the limit on how long to wait for an input operation to complete. If a receive operation has blocked for this much time without receiving additional data, it shall return with a partial count or <i>errno</i> set to [EAGAIN] or [EWOULDBLOCK] if no data is received. The default for this option is zero, which indicates that a receive operation shall not time out. This option takes a timeval structure. Note that not all implementations allow this option to be set.	
40695			
40696			
40697			
40698			
40699			
40700			
40701			
40702			
40703			
40704	SO_SNDLOWAT	Sets the minimum number of bytes to process for socket output operations. Non-blocking output operations shall process no data if flow control does not allow the smaller of the send low water mark value or the entire request to be processed. This option takes an int value. Note that not all implementations allow this option to be set.	
40705			
40706			
40707			
40708			
40709	SO_SNDTIMEO	Sets the timeout value specifying the amount of time that an output function blocks because flow control prevents data from being sent. If a send operation has blocked for this time, it shall return with a partial count or with <i>errno</i> set to [EAGAIN] or [EWOULDBLOCK] if no data is sent. The default for this option is zero, which indicates that a send operation shall not time out. This option stores a timeval structure. Note that not all implementations allow this option to be set.	
40710			
40711			
40712			
40713			
40714			
40715			
40716	For Boolean options, 0 indicates that the option is disabled and 1 indicates that the option is enabled.		
40717			
40718	Options at other protocol levels vary in format and name.		
40719	RETURN VALUE		
40720	Upon successful completion, <i>setsockopt()</i> shall return 0. Otherwise, -1 shall be returned and <i>errno</i> set to indicate the error.		
40721			
40722	ERRORS		
40723	The <i>setsockopt()</i> function shall fail if:		
40724	[EBADF]	The <i>socket</i> argument is not a valid file descriptor.	
40725	[EDOM]	The send and receive timeout values are too big to fit into the timeout fields in the socket structure.	
40726			

40727	[EINVAL]	The specified option is invalid at the specified socket level or the socket has been shut down.
40728		
40729	[EISCONN]	The socket is already connected, and a specified option cannot be set while the socket is connected.
40730		
40731	[ENOPROTOOPT]	
40732		The option is not supported by the protocol.
40733	[ENOTSOCK]	The <i>socket</i> argument does not refer to a socket.
40734		The <i>setsockopt()</i> function may fail if:
40735	[ENOMEM]	There was insufficient memory available for the operation to complete.
40736	[ENOBUFS]	Insufficient resources are available in the system to complete the call.
40737	EXAMPLES	
40738	None.	
40739	APPLICATION USAGE	
40740	The <i>setsockopt()</i> function provides an application program with the means to control socket behavior. An application program can use <i>setsockopt()</i> to allocate buffer space, control timeouts, or permit socket data broadcasts. The <code><sys/socket.h></code> header defines the socket-level options available to <i>setsockopt()</i> .	
40741		
40742		
40743		
40744	Options may exist at multiple protocol levels. The <code>SO_</code> options are always present at the uppermost socket level.	
40745		
40746	RATIONALE	
40747	None.	
40748	FUTURE DIRECTIONS	
40749	None.	
40750	SEE ALSO	
40751	Section 2.10 (on page 58), <i>bind()</i> , <i>endprotoent()</i> , <i>getsockopt()</i> , <i>socket()</i> , the Base Definitions volume of IEEE Std 1003.1-2001, <code><netinet/in.h></code> , <code><sys/socket.h></code>	
40752		
40753	CHANGE HISTORY	
40754	First released in Issue 6. Derived from the XNS, Issue 5.2 specification.	

40755 NAME

40756 setstate — switch pseudo-random number generator state arrays

40757 SYNOPSIS

40758 xSI #include <stdlib.h>

40759 char *setstate(const char *state);

40760

40761 DESCRIPTION

40762 Refer to *initstate()*.

40763 **NAME**

40764 setuid — set user ID

40765 **SYNOPSIS**

40766 #include <unistd.h>

40767 int setuid(uid_t uid);

40768 **DESCRIPTION**40769 If the process has appropriate privileges, *setuid()* shall set the real user ID, effective user ID, and
40770 the saved set-user-ID of the calling process to *uid*.40771 If the process does not have appropriate privileges, but *uid* is equal to the real user ID or the
40772 saved set-user-ID, *setuid()* shall set the effective user ID to *uid*; the real user ID and saved set-
40773 user-ID shall remain unchanged.40774 The *setuid()* function shall not affect the supplementary group list in any way.40775 **RETURN VALUE**40776 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
40777 indicate the error.40778 **ERRORS**40779 The *setuid()* function shall fail, return -1, and set *errno* to the corresponding value if one or more
40780 of the following are true:40781 [EINVAL] The value of the *uid* argument is invalid and not supported by the
40782 implementation.40783 [EPERM] The process does not have appropriate privileges and *uid* does not match the
40784 real user ID or the saved set-user-ID.40785 **EXAMPLES**

40786 None.

40787 **APPLICATION USAGE**

40788 None.

40789 **RATIONALE**40790 The various behaviors of the *setuid()* and *setgid()* functions when called by non-privileged
40791 processes reflect the behavior of different historical implementations. For portability, it is
40792 recommended that new non-privileged applications use the *seteuid()* and *setegid()* functions
40793 instead.40794 The saved set-user-ID capability allows a program to regain the effective user ID established at
40795 the last *exec* call. Similarly, the saved set-group-ID capability allows a program to regain the
40796 effective group ID established at the last *exec* call. These capabilities are derived from System V.
40797 Without them, a program might have to run as superuser in order to perform the same
40798 functions, because superuser can write on the user's files. This is a problem because such a
40799 program can write on any user's files, and so must be carefully written to emulate the
40800 permissions of the calling process properly. In System V, these capabilities have traditionally
40801 been implemented only via the *setuid()* and *setgid()* functions for non-privileged processes. The
40802 fact that the behavior of those functions was different for privileged processes made them
40803 difficult to use. The POSIX.1-1990 standard defined the *setuid()* function to behave differently
40804 for privileged and unprivileged users. When the caller had the appropriate privilege, the
40805 function set the calling process' real user ID, effective user ID, and saved set-user ID on
40806 implementations that supported it. When the caller did not have the appropriate privilege, the
40807 function set only the effective user ID, subject to permission checks. The former use is generally
40808 needed for utilities like *login* and *su*, which are not conforming applications and thus outside the

scope of IEEE Std 1003.1-2001. These utilities wish to change the user ID irrevocably to a new value, generally that of an unprivileged user. The latter use is needed for conforming applications that are installed with the set-user-ID bit and need to perform operations using the real user ID.

IEEE Std 1003.1-2001 augments the latter functionality with a mandatory feature named `_POSIX_SAVED_IDS`. This feature permits a set-user-ID application to switch its effective user ID back and forth between the values of its *exec*-time real user ID and effective user ID. Unfortunately, the POSIX.1-1990 standard did not permit a conforming application using this feature to work properly when it happened to be executed with the (implementation-defined) appropriate privilege. Furthermore, the application did not even have a means to tell whether it had this privilege. Since the saved set-user-ID feature is quite desirable for applications, as evidenced by the fact that NIST required it in FIPS 151-2, it has been mandated by IEEE Std 1003.1-2001. However, there are implementors who have been reluctant to support it given the limitation described above.

The 4.3BSD system handles the problem by supporting separate functions: *setuid()* (which always sets both the real and effective user IDs, like *setuid()* in IEEE Std 1003.1-2001 for privileged users), and *seteuid()* (which always sets just the effective user ID, like *setuid()* in IEEE Std 1003.1-2001 for non-privileged users). This separation of functionality into distinct functions seems desirable. 4.3BSD does not support the saved set-user-ID feature. It supports similar functionality of switching the effective user ID back and forth via *setreuid()*, which permits reversing the real and effective user IDs. This model seems less desirable than the saved set-user-ID because the real user ID changes as a side effect. The current 4.4BSD includes saved effective IDs and uses them for *seteuid()* and *setegid()* as described above. The *setreuid()* and *setregid()* functions will be deprecated or removed.

The solution here is:

- Require that all implementations support the functionality of the saved set-user-ID, which is set by the *exec* functions and by privileged calls to *setuid()*.
- Add the *seteuid()* and *setegid()* functions as portable alternatives to *setuid()* and *setgid()* for non-privileged and privileged processes.

Historical systems have provided two mechanisms for a set-user-ID process to change its effective user ID to be the same as its real user ID in such a way that it could return to the original effective user ID: the use of the *setuid()* function in the presence of a saved set-user-ID, or the use of the BSD *setreuid()* function, which was able to swap the real and effective user IDs. The changes included in IEEE Std 1003.1-2001 provide a new mechanism using *seteuid()* in conjunction with a saved set-user-ID. Thus, all implementations with the new *seteuid()* mechanism will have a saved set-user-ID for each process, and most of the behavior controlled by `_POSIX_SAVED_IDS` has been changed to agree with the case where the option was defined. The *kill()* function is an exception. Implementors of the new *seteuid()* mechanism will generally be required to maintain compatibility with the older mechanisms previously supported by their systems. However, compatibility with this use of *setreuid()* and with the `_POSIX_SAVED_IDS` behavior of *kill()* is unfortunately complicated. If an implementation with a saved set-user-ID allows a process to use *setreuid()* to swap its real and effective user IDs, but were to leave the saved set-user-ID unmodified, the process would then have an effective user ID equal to the original real user ID, and both real and saved set-user-ID would be equal to the original effective user ID. In that state, the real user would be unable to kill the process, even though the effective user ID of the process matches that of the real user, if the *kill()* behavior of `_POSIX_SAVED_IDS` was used. This is obviously not acceptable. The alternative choice, which is used in at least one implementation, is to change the saved set-user-ID to the effective user ID during most calls to *setreuid()*. The standard developers considered that alternative to be less correct than the

40858 retention of the old behavior of *kill()* in such systems. Current conforming applications shall
40859 accommodate either behavior from *kill()*, and there appears to be no strong reason for *kill()* to
40860 check the saved set-user-ID rather than the effective user ID.

40861 FUTURE DIRECTIONS

40862 None.

40863 SEE ALSO

40864 *exec*, *getegid()*, *geteuid()*, *getgid()*, *getuid()*, *setegid()*, *seteuid()*, *setgid()*, *setregid()*, *setreuid()*, the
40865 Base Definitions volume of IEEE Std 1003.1-2001, **<sys/types.h>**, **<unistd.h>**

40866 CHANGE HISTORY

40867 First released in Issue 1. Derived from Issue 1 of the SVID.

40868 Issue 6

40869 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

40870 The following new requirements on POSIX implementations derive from alignment with the
40871 Single UNIX Specification:

- 40872 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
40873 required for conforming implementations of previous POSIX specifications, it was not
40874 required for UNIX applications.
- 40875 • The functionality associated with `_POSIX_SAVED_IDS` is now mandatory. This is a FIPS
40876 requirement.

40877 The following changes were made to align with the IEEE P1003.1a draft standard:

- 40878 • The effects of *setuid()* in processes without appropriate privileges are changed.
- 40879 • A requirement that the supplementary group list is not affected is added.

40880 NAME

40881 setutxent — reset the user accounting database to the first entry

40882 SYNOPSIS

40883 XSI #include <utmpx.h>

40884 void setutxent(void);

40885

40886 DESCRIPTION

40887 Refer to *endutxent()*.

40888 **NAME**

40889 setvbuf — assign buffering to a stream

40890 **SYNOPSIS**

40891 #include <stdio.h>

```
40892       int setvbuf(FILE *restrict stream, char *restrict buf, int type,
40893                   size_t size);
```

40894 **DESCRIPTION**

40895 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 40896 conflict between the requirements described here and the ISO C standard is unintentional. This
 40897 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

40898 The *setvbuf()* function may be used after the stream pointed to by *stream* is associated with an
 40899 open file but before any other operation (other than an unsuccessful call to *setvbuf()*) is
 40900 performed on the stream. The argument *type* determines how *stream* shall be buffered, as
 40901 follows:

- 40902 • {_IOFBF} shall cause input/output to be fully buffered.
- 40903 • {_IOLBF} shall cause input/output to be line buffered.
- 40904 • {_IONBF} shall cause input/output to be unbuffered.

40905 If *buf* is not a null pointer, the array it points to may be used instead of a buffer allocated by
 40906 *setvbuf()* and the argument *size* specifies the size of the array; otherwise, *size* may determine the
 40907 size of a buffer allocated by the *setvbuf()* function. The contents of the array at any time are
 40908 unspecified.

40909 For information about streams, see Section 2.5 (on page 34).

40910 **RETURN VALUE**

40911 Upon successful completion, *setvbuf()* shall return 0. Otherwise, it shall return a non-zero value
 40912 CX if an invalid value is given for *type* or if the request cannot be honored, and may set *errno* to
 40913 indicate the error.

40914 **ERRORS**

40915 The *setvbuf()* function may fail if:

40916 CX [EBADF] The file descriptor underlying *stream* is not valid.

40917 **EXAMPLES**

40918 None.

40919 **APPLICATION USAGE**

40920 A common source of error is allocating buffer space as an “automatic” variable in a code block,
 40921 and then failing to close the stream in the same block.

40922 With *setvbuf()*, allocating a buffer of *size* bytes does not necessarily imply that all of *size* bytes are
 40923 used for the buffer area.

40924 Applications should note that many implementations only provide line buffering on input from
 40925 terminal devices.

40926 **RATIONALE**

40927 None.

40928 FUTURE DIRECTIONS

40929 None.

40930 SEE ALSO

40931 Section 2.5 (on page 34), *fopen()*, *setbuf()*, the Base Definitions volume of IEEE Std 1003.1-2001,
40932 <**stdio.h**>

40933 CHANGE HISTORY

40934 First released in Issue 1. Derived from Issue 1 of the SVID.

40935 Issue 6

40936 Extensions beyond the ISO C standard are marked.

40937 The *setvbuf()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

40938 **NAME**40939 shm_open — open a shared memory object (**REALTIME**)40940 **SYNOPSIS**

40941 SHM #include <sys/mman.h>

40942 int shm_open(const char *name, int oflag, mode_t mode);

40943

40944 **DESCRIPTION**

40945 The *shm_open()* function shall establish a connection between a shared memory object and a file
 40946 descriptor. It shall create an open file description that refers to the shared memory object and a
 40947 file descriptor that refers to that open file description. The file descriptor is used by other
 40948 functions to refer to that shared memory object. The *name* argument points to a string naming a
 40949 shared memory object. It is unspecified whether the name appears in the file system and is
 40950 visible to other functions that take pathnames as arguments. The *name* argument conforms to the
 40951 construction rules for a pathname. If *name* begins with the slash character, then processes calling
 40952 *shm_open()* with the same value of *name* refer to the same shared memory object, as long as that
 40953 name has not been removed. If *name* does not begin with the slash character, the effect is
 40954 implementation-defined. The interpretation of slash characters other than the leading slash
 40955 character in *name* is implementation-defined.

40956 If successful, *shm_open()* shall return a file descriptor for the shared memory object that is the
 40957 lowest numbered file descriptor not currently open for that process. The open file description is
 40958 new, and therefore the file descriptor does not share it with any other processes. It is unspecified
 40959 whether the file offset is set. The FD_CLOEXEC file descriptor flag associated with the new file
 40960 descriptor is set.

40961 The file status flags and file access modes of the open file description are according to the value
 40962 of *oflag*. The *oflag* argument is the bitwise-inclusive OR of the following flags defined in the
 40963 <fcntl.h> header. Applications specify exactly one of the first two values (access modes) below
 40964 in the value of *oflag*:

40965 O_RDONLY Open for read access only.

40966 O_RDWR Open for read or write access.

40967 Any combination of the remaining flags may be specified in the value of *oflag*:

40968 O_CREAT If the shared memory object exists, this flag has no effect, except as noted
 40969 under O_EXCL below. Otherwise, the shared memory object is created; the
 40970 user ID of the shared memory object shall be set to the effective user ID of the
 40971 process; the group ID of the shared memory object is set to a system default
 40972 group ID or to the effective group ID of the process. The permission bits of the
 40973 shared memory object shall be set to the value of the *mode* argument except
 40974 those set in the file mode creation mask of the process. When bits in *mode*
 40975 other than the file permission bits are set, the effect is unspecified. The *mode*
 40976 argument does not affect whether the shared memory object is opened for
 40977 reading, for writing, or for both. The shared memory object has a size of zero.

40978 O_EXCL If O_EXCL and O_CREAT are set, *shm_open()* fails if the shared memory
 40979 object exists. The check for the existence of the shared memory object and the
 40980 creation of the object if it does not exist is atomic with respect to other
 40981 processes executing *shm_open()* naming the same shared memory object with
 40982 O_EXCL and O_CREAT set. If O_EXCL is set and O_CREAT is not set, the
 40983 result is undefined.

40984 O_TRUNC If the shared memory object exists, and it is successfully opened O_RDWR,
 40985 the object shall be truncated to zero length and the mode and owner shall be
 40986 unchanged by this function call. The result of using O_TRUNC with
 40987 O_RDONLY is undefined.

40988 When a shared memory object is created, the state of the shared memory object, including all
 40989 data associated with the shared memory object, persists until the shared memory object is
 40990 unlinked and all other references are gone. It is unspecified whether the name and shared
 40991 memory object state remain valid after a system reboot.

40992 RETURN VALUE

40993 Upon successful completion, the *shm_open()* function shall return a non-negative integer
 40994 representing the lowest numbered unused file descriptor. Otherwise, it shall return -1 and set
 40995 *errno* to indicate the error.

40996 ERRORS

40997 The *shm_open()* function shall fail if:

40998 [EACCES] The shared memory object exists and the permissions specified by *oflag* are
 40999 denied, or the shared memory object does not exist and permission to create
 41000 the shared memory object is denied, or O_TRUNC is specified and write
 41001 permission is denied.

41002 [EEXIST] O_CREAT and O_EXCL are set and the named shared memory object already
 41003 exists.

41004 [EINTR] The *shm_open()* operation was interrupted by a signal.

41005 [EINVAL] The *shm_open()* operation is not supported for the given name.

41006 [EMFILE] Too many file descriptors are currently in use by this process.

41007 [ENAMETOOLONG]

41008 The length of the *name* argument exceeds {PATH_MAX} or a pathname
 41009 component is longer than {NAME_MAX}.

41010 [ENFILE] Too many shared memory objects are currently open in the system.

41011 [ENOENT] O_CREAT is not set and the named shared memory object does not exist.

41012 [ENOSPC] There is insufficient space for the creation of the new shared memory object.

41013 EXAMPLES

41014 None.

41015 APPLICATION USAGE

41016 None.

41017 RATIONALE

41018 When the Memory Mapped Files option is supported, the normal *open()* call is used to obtain a
 41019 descriptor to a file to be mapped according to existing practice with *mmap()*. When the Shared
 41020 Memory Objects option is supported, the *shm_open()* function shall obtain a descriptor to the
 41021 shared memory object to be mapped.

41022 There is ample precedent for having a file descriptor represent several types of objects. In the
 41023 POSIX.1-1990 standard, a file descriptor can represent a file, a pipe, a FIFO, a tty, or a directory.
 41024 Many implementations simply have an operations vector, which is indexed by the file descriptor
 41025 type and does very different operations. Note that in some cases the file descriptor passed to
 41026 generic operations on file descriptors is returned by *open()* or *creat()* and in some cases returned
 41027 by alternate functions, such as *pipe()*. The latter technique is used by *shm_open()*.

Note that such shared memory objects can actually be implemented as mapped files. In both cases, the size can be set after the open using *ftruncate()*. The *shm_open()* function itself does not create a shared object of a specified size because this would duplicate an extant function that set the size of an object referenced by a file descriptor.

On implementations where memory objects are implemented using the existing file system, the *shm_open()* function may be implemented using a macro that invokes *open()*, and the *shm_unlink()* function may be implemented using a macro that invokes *unlink()*.

For implementations without a permanent file system, the definition of the name of the memory objects is allowed not to survive a system reboot. Note that this allows systems with a permanent file system to implement memory objects as data structures internal to the implementation as well.

On implementations that choose to implement memory objects using memory directly, a *shm_open()* followed by an *ftruncate()* and *close()* can be used to preallocate a shared memory area and to set the size of that preallocation. This may be necessary for systems without virtual memory hardware support in order to ensure that the memory is contiguous.

The set of valid open flags to *shm_open()* was restricted to *O_RDONLY*, *O_RDWR*, *O_CREAT*, and *O_TRUNC* because these could be easily implemented on most memory mapping systems. This volume of IEEE Std 1003.1-2001 is silent on the results if the implementation cannot supply the requested file access because of implementation-defined reasons, including hardware ones.

The error conditions [EACCES] and [ENOTSUP] are provided to inform the application that the implementation cannot complete a request.

[EACCES] indicates for implementation-defined reasons, probably hardware-related, that the implementation cannot comply with a requested mode because it conflicts with another requested mode. An example might be that an application desires to open a memory object two times, mapping different areas with different access modes. If the implementation cannot map a single area into a process space in two places, which would be required if different access modes were required for the two areas, then the implementation may inform the application at the time of the second open.

[ENOTSUP] indicates for implementation-defined reasons, probably hardware-related, that the implementation cannot comply with a requested mode at all. An example would be that the hardware of the implementation cannot support write-only shared memory areas.

On all implementations, it may be desirable to restrict the location of the memory objects to specific file systems for performance (such as a RAM disk) or implementation-defined reasons (shared memory supported directly only on certain file systems). The *shm_open()* function may be used to enforce these restrictions. There are a number of methods available to the application to determine an appropriate name of the file or the location of an appropriate directory. One way is from the environment via *getenv()*. Another would be from a configuration file.

This volume of IEEE Std 1003.1-2001 specifies that memory objects have initial contents of zero when created. This is consistent with current behavior for both files and newly allocated memory. For those implementations that use physical memory, it would be possible that such implementations could simply use available memory and give it to the process uninitialized. This, however, is not consistent with standard behavior for the uninitialized data area, the stack, and of course, files. Finally, it is highly desirable to set the allocated memory to zero for security reasons. Thus, initializing memory objects to zero is required.

41072 FUTURE DIRECTIONS

41073 None.

41074 SEE ALSO

41075 *close()*, *dup()*, *exec*, *fcntl()*, *mmap()*, *shmat()*, *shmctl()*, *shmdt()*, *shm_unlink()*, *umask()*, the Base
41076 Definitions volume of IEEE Std 1003.1-2001, **<fcntl.h>**, **<sys/mman.h>**

41077 CHANGE HISTORY

41078 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

41079 Issue 6

41080 The *shm_open()* function is marked as part of the Shared Memory Objects option.

41081 The [ENOSYS] error condition has been removed as stubs need not be provided if an
41082 implementation does not support the Shared Memory Objects option.

41083 **NAME**41084 shm_unlink — remove a shared memory object (**REALTIME**)41085 **SYNOPSIS**

41086 SHM #include <sys/mman.h>

41087 int shm_unlink(const char *name);

41088

41089 **DESCRIPTION**41090 The *shm_unlink()* function shall remove the name of the shared memory object named by the
41091 string pointed to by *name*.41092 If one or more references to the shared memory object exist when the object is unlinked, the
41093 name shall be removed before *shm_unlink()* returns, but the removal of the memory object
41094 contents shall be postponed until all open and map references to the shared memory object have
41095 been removed.41096 Even if the object continues to exist after the last *shm_unlink()*, reuse of the name shall
41097 subsequently cause *shm_open()* to behave as if no shared memory object of this name exists (that
41098 is, *shm_open()* will fail if O_CREAT is not set, or will create a new shared memory object if
41099 O_CREAT is set).41100 **RETURN VALUE**41101 Upon successful completion, a value of zero shall be returned. Otherwise, a value of -1 shall be
41102 returned and *errno* set to indicate the error. If -1 is returned, the named shared memory object
41103 shall not be changed by this function call.41104 **ERRORS**41105 The *shm_unlink()* function shall fail if:

41106 [EACCES] Permission is denied to unlink the named shared memory object.

41107 [ENAMETOOLONG]

41108 The length of the *name* argument exceeds {PATH_MAX} or a pathname
41109 component is longer than {NAME_MAX}.

41110 [ENOENT] The named shared memory object does not exist.

41111 **EXAMPLES**

41112 None.

41113 **APPLICATION USAGE**41114 Names of memory objects that were allocated with *open()* are deleted with *unlink()* in the usual
41115 fashion. Names of memory objects that were allocated with *shm_open()* are deleted with
41116 *shm_unlink()*. Note that the actual memory object is not destroyed until the last close and
41117 unmap on it have occurred if it was already in use.41118 **RATIONALE**

41119 None.

41120 **FUTURE DIRECTIONS**

41121 None.

41122 **SEE ALSO**41123 *close()*, *mmap()*, *munmap()*, *shmat()*, *shmctl()*, *shmdt()*, *shm_open()*, the Base Definitions volume
41124 of IEEE Std 1003.1-2001, <sys/mman.h>

41125 CHANGE HISTORY

41126 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

41127 Issue 6

41128 The *shm_unlink()* function is marked as part of the Shared Memory Objects option.

41129 In the DESCRIPTION, text is added to clarify that reusing the same name after a *shm_unlink()*
41130 will not attach to the old shared memory object.

41131 The [ENOSYS] error condition has been removed as stubs need not be provided if an
41132 implementation does not support the Shared Memory Objects option.

41133 **NAME**41134 `shmat` — XSI shared memory attach operation41135 **SYNOPSIS**41136 XSI `#include <sys/shm.h>`41137 `void *shmat(int shmid, const void *shmaddr, int shmflg);`

41138

41139 **DESCRIPTION**

41140 The `shmat()` function operates on XSI shared memory (see the Base Definitions volume of
 41141 IEEE Std 1003.1-2001, Section 3.340, Shared Memory Object). It is unspecified whether this
 41142 function interoperates with the realtime interprocess communication facilities defined in Section
 41143 2.8 (on page 41).

41144 The `shmat()` function attaches the shared memory segment associated with the shared memory
 41145 identifier specified by *shmid* to the address space of the calling process. The segment is attached
 41146 at the address specified by one of the following criteria:

- 41147 • If *shmaddr* is a null pointer, the segment is attached at the first available address as selected
 41148 by the system.
- 41149 • If *shmaddr* is not a null pointer and (*shmflg* & SHM_RND) is non-zero, the segment is attached
 41150 at the address given by (*shmaddr* - ((*uintptr_t*)*shmaddr* % SHMLBA)). The character '`%`' is the
 41151 C-language remainder operator.
- 41152 • If *shmaddr* is not a null pointer and (*shmflg* & SHM_RND) is 0, the segment is attached at the
 41153 address given by *shmaddr*.
- 41154 • The segment is attached for reading if (*shmflg* & SHM_RDONLY) is non-zero and the calling
 41155 process has read permission; otherwise, if it is 0 and the calling process has read and write
 41156 permission, the segment is attached for reading and writing.

41157 **RETURN VALUE**

41158 Upon successful completion, `shmat()` shall increment the value of *shm_nattch* in the data
 41159 structure associated with the shared memory ID of the attached shared memory segment and
 41160 return the segment's start address.

41161 Otherwise, the shared memory segment shall not be attached, `shmat()` shall return `-1`, and *errno*
 41162 shall be set to indicate the error.

41163 **ERRORS**41164 The `shmat()` function shall fail if:

- | | | |
|---|----------|--|
| 41165
41166 | [EACCES] | Operation permission is denied to the calling process; see Section 2.7 (on page 39). |
| 41167
41168
41169
41170
41171 | [EINVAL] | The value of <i>shmid</i> is not a valid shared memory identifier, the <i>shmaddr</i> is not a null pointer, and the value of (<i>shmaddr</i> - ((<i>uintptr_t</i>) <i>shmaddr</i> % SHMLBA)) is an illegal address for attaching shared memory; or the <i>shmaddr</i> is not a null pointer, (<i>shmflg</i> & SHM_RND) is 0, and the value of <i>shmaddr</i> is an illegal address for attaching shared memory. |
| 41172
41173 | [EMFILE] | The number of shared memory segments attached to the calling process would exceed the system-imposed limit. |
| 41174
41175 | [ENOMEM] | The available data space is not large enough to accommodate the shared memory segment. |

41176 **EXAMPLES**

41177 None.

41178 **APPLICATION USAGE**

41179 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
41180 Application developers who need to use IPC should design their applications so that modules
41181 using the IPC routines described in Section 2.7 (on page 39) can be easily modified to use the
41182 alternative interfaces.

41183 **RATIONALE**

41184 None.

41185 **FUTURE DIRECTIONS**

41186 None.

41187 **SEE ALSO**

41188 Section 2.7 (on page 39), Section 2.8 (on page 41), *exec*, *exit()*, *fork()*, *shmctl()*, *shmdt()*, *shmget()*,
41189 *shm_open()*, *shm_unlink()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/shm.h>

41190 **CHANGE HISTORY**

41191 First released in Issue 2. Derived from Issue 2 of the SVID.

41192 **Issue 5**

41193 Moved from SHARED MEMORY to BASE.

41194 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
41195 DIRECTIONS to a new APPLICATION USAGE section.

41196 **Issue 6**

41197 The Open Group Corrigendum U021/13 is applied.

41198 **NAME**

41199 shmctl — XSI shared memory control operations

41200 **SYNOPSIS**

41201 XSI #include <sys/shm.h>

41202 int shmctl(int *shmid*, int *cmd*, struct shm_id_ds **buf*);

41203

41204 **DESCRIPTION**

41205 The *shmctl()* function operates on XSI shared memory (see the Base Definitions volume of
 41206 IEEE Std 1003.1-2001, Section 3.340, Shared Memory Object). It is unspecified whether this
 41207 function interoperates with the realtime interprocess communication facilities defined in Section
 41208 2.8 (on page 41).

41209 The *shmctl()* function provides a variety of shared memory control operations as specified by
 41210 *cmd*. The following values for *cmd* are available:

41211 IPC_STAT Place the current value of each member of the **shm_id_ds** data structure
 41212 associated with *shmid* into the structure pointed to by *buf*. The contents of the
 41213 structure are defined in <sys/shm.h>.

41214 IPC_SET Set the value of the following members of the **shm_id_ds** data structure
 41215 associated with *shmid* to the corresponding value found in the structure
 41216 pointed to by *buf*:

41217 shm_perm.uid
 41218 shm_perm.gid
 41219 shm_perm.mode Low-order nine bits.

41220 IPC_SET can only be executed by a process that has an effective user ID equal
 41221 to either that of a process with appropriate privileges or to the value of
 41222 *shm_perm.cuid* or *shm_perm.uid* in the **shm_id_ds** data structure associated with
 41223 *shmid*.

41224 IPC_RMID Remove the shared memory identifier specified by *shmid* from the system and
 41225 destroy the shared memory segment and **shm_id_ds** data structure associated
 41226 with it. IPC_RMID can only be executed by a process that has an effective user
 41227 ID equal to either that of a process with appropriate privileges or to the value
 41228 of *shm_perm.cuid* or *shm_perm.uid* in the **shm_id_ds** data structure associated
 41229 with *shmid*.

41230 **RETURN VALUE**

41231 Upon successful completion, *shmctl()* shall return 0; otherwise, it shall return -1 and set *errno* to
 41232 indicate the error.

41233 **ERRORS**41234 The *shmctl()* function shall fail if:

41235 [EACCES] The argument *cmd* is equal to IPC_STAT and the calling process does not have
 41236 read permission; see Section 2.7 (on page 39).

41237 [EINVAL] The value of *shmid* is not a valid shared memory identifier, or the value of *cmd*
 41238 is not a valid command.

41239 [EPERM] The argument *cmd* is equal to IPC_RMID or IPC_SET and the effective user ID
 41240 of the calling process is not equal to that of a process with appropriate
 41241 privileges and it is not equal to the value of *shm_perm.cuid* or *shm_perm.uid* in
 41242 the data structure associated with *shmid*.

41243 The *shmctl()* function may fail if:

41244 [EOVERFLOW] The *cmd* argument is IPC_STAT and the *gid* or *uid* value is too large to be
41245 stored in the structure pointed to by the *buf* argument.

41246 EXAMPLES

41247 None.

41248 APPLICATION USAGE

41249 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
41250 Application developers who need to use IPC should design their applications so that modules
41251 using the IPC routines described in Section 2.7 (on page 39) can be easily modified to use the
41252 alternative interfaces.

41253 RATIONALE

41254 None.

41255 FUTURE DIRECTIONS

41256 None.

41257 SEE ALSO

41258 Section 2.7 (on page 39), Section 2.8 (on page 41), *shmat()*, *shmdt()*, *shmget()*, *shm_open()*,
41259 *shm_unlink()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/shm.h>

41260 CHANGE HISTORY

41261 First released in Issue 2. Derived from Issue 2 of the SVID.

41262 Issue 5

41263 Moved from SHARED MEMORY to BASE.

41264 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
41265 DIRECTIONS to a new APPLICATION USAGE section.

41266 **NAME**

41267 shmdt — XSI shared memory detach operation

41268 **SYNOPSIS**

41269 XSI #include <sys/shm.h>

41270 int shmdt(const void *shmaddr);

41271

41272 **DESCRIPTION**

41273 The *shmdt()* function operates on XSI shared memory (see the Base Definitions volume of
 41274 IEEE Std 1003.1-2001, Section 3.340, Shared Memory Object). It is unspecified whether this
 41275 function interoperates with the realtime interprocess communication facilities defined in Section
 41276 2.8 (on page 41).

41277 The *shmdt()* function detaches the shared memory segment located at the address specified by
 41278 *shmaddr* from the address space of the calling process.

41279 **RETURN VALUE**

41280 Upon successful completion, *shmdt()* shall decrement the value of *shm_nattch* in the data
 41281 structure associated with the shared memory ID of the attached shared memory segment and
 41282 return 0.

41283 Otherwise, the shared memory segment shall not be detached, *shmdt()* shall return *-1*, and *errno*
 41284 shall be set to indicate the error.

41285 **ERRORS**41286 The *shmdt()* function shall fail if:

41287 [EINVAL] The value of *shmaddr* is not the data segment start address of a shared
 41288 memory segment.

41289 **EXAMPLES**

41290 None.

41291 **APPLICATION USAGE**

41292 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
 41293 Application developers who need to use IPC should design their applications so that modules
 41294 using the IPC routines described in Section 2.7 (on page 39) can be easily modified to use the
 41295 alternative interfaces.

41296 **RATIONALE**

41297 None.

41298 **FUTURE DIRECTIONS**

41299 None.

41300 **SEE ALSO**

41301 Section 2.7 (on page 39), Section 2.8 (on page 41), *exec*, *exit()*, *fork()*, *shmat()*, *shmctl()*, *shmget()*,
 41302 *shm_open()*, *shm_unlink()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/shm.h>

41303 **CHANGE HISTORY**

41304 First released in Issue 2. Derived from Issue 2 of the SVID.

41305 **Issue 5**

41306 Moved from SHARED MEMORY to BASE.

41307 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
 41308 DIRECTIONS to a new APPLICATION USAGE section.

41309 NAME

41310 shmget — get an XSI shared memory segment

41311 SYNOPSIS

41312 XSI

```
#include <sys/shm.h>
```

41313

```
int shmget(key_t key, size_t size, int shmflg);
```

41314

41315 DESCRIPTION

41316 The *shmget()* function operates on XSI shared memory (see the Base Definitions volume of
 41317 IEEE Std 1003.1-2001, Section 3.340, Shared Memory Object). It is unspecified whether this
 41318 function interoperates with the realtime interprocess communication facilities defined in Section
 41319 2.8 (on page 41).

41320 The *shmget()* function shall return the shared memory identifier associated with *key*.

41321 A shared memory identifier, associated data structure, and shared memory segment of at least
 41322 *size* bytes (see <sys/shm.h>) are created for *key* if one of the following is true:

- 41323 • The argument *key* is equal to IPC_PRIVATE.
- 41324 • The argument *key* does not already have a shared memory identifier associated with it and
 41325 (*shmflg* & IPC_CREAT) is non-zero.

41326 Upon creation, the data structure associated with the new shared memory identifier shall be
 41327 initialized as follows:

- 41328 • The values of *shm_perm.cuid*, *shm_perm.uid*, *shm_perm.cgid*, and *shm_perm.gid* are set equal to
 41329 the effective user ID and effective group ID, respectively, of the calling process.
- 41330 • The low-order nine bits of *shm_perm.mode* are set equal to the low-order nine bits of *shmflg*.
- 41331 • The value of *shm_segsz* is set equal to the value of *size*.
- 41332 • The values of *shm_lpid*, *shm_nattch*, *shm_atime*, and *shm_dtime* are set equal to 0.
- 41333 • The value of *shm_ctime* is set equal to the current time.

41334 When the shared memory segment is created, it shall be initialized with all zero values.

41335 RETURN VALUE

41336 Upon successful completion, *shmget()* shall return a non-negative integer, namely a shared
 41337 memory identifier; otherwise, it shall return -1 and set *errno* to indicate the error.

41338 ERRORS

41339 The *shmget()* function shall fail if:

- | | | |
|-------------------------|----------|---|
| 41340
41341
41342 | [EACCES] | A shared memory identifier exists for <i>key</i> but operation permission as specified by the low-order nine bits of <i>shmflg</i> would not be granted; see Section 2.7 (on page 39). |
| 41343
41344 | [EEXIST] | A shared memory identifier exists for the argument <i>key</i> but (<i>shmflg</i> & IPC_CREAT) && (<i>shmflg</i> & IPC_EXCL) is non-zero. |
| 41345
41346 | [EINVAL] | A shared memory segment is to be created and the value of <i>size</i> is less than the system-imposed minimum or greater than the system-imposed maximum. |
| 41347
41348
41349 | [EINVAL] | No shared memory segment is to be created and a shared memory segment exists for <i>key</i> but the size of the segment associated with it is less than <i>size</i> and <i>size</i> is not 0. |

41350 [ENOENT] A shared memory identifier does not exist for the argument *key* and (*shmflg*
41351 &IPC_CREAT) is 0.

41352 [ENOMEM] A shared memory identifier and associated shared memory segment shall be
41353 created, but the amount of available physical memory is not sufficient to fill
41354 the request.

41355 [ENOSPC] A shared memory identifier is to be created, but the system-imposed limit on
41356 the maximum number of allowed shared memory identifiers system-wide
41357 would be exceeded.

41358 EXAMPLES

41359 None.

41360 APPLICATION USAGE

41361 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
41362 Application developers who need to use IPC should design their applications so that modules
41363 using the IPC routines described in Section 2.7 (on page 39) can be easily modified to use the
41364 alternative interfaces.

41365 RATIONALE

41366 None.

41367 FUTURE DIRECTIONS

41368 None.

41369 SEE ALSO

41370 Section 2.7 (on page 39), Section 2.8 (on page 41), *shmat()*, *shmctl()*, *shmdt()*, *shm_open()*,
41371 *shm_unlink()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/shm.h>

41372 CHANGE HISTORY

41373 First released in Issue 2. Derived from Issue 2 of the SVID.

41374 Issue 5

41375 Moved from SHARED MEMORY to BASE.

41376 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
41377 DIRECTIONS to a new APPLICATION USAGE section.

41378 **NAME**

41379 shutdown — shut down socket send and receive operations

41380 **SYNOPSIS**

41381 #include <sys/socket.h>

41382 int shutdown(int *socket*, int *how*);41383 **DESCRIPTION**41384 The *shutdown()* function shall cause all or part of a full-duplex connection on the socket
41385 associated with the file descriptor *socket* to be shut down.41386 The *shutdown()* function takes the following arguments:41387 *socket* Specifies the file descriptor of the socket.41388 *how* Specifies the type of shutdown. The values are as follows:

41389 SHUT_RD Disables further receive operations.

41390 SHUT_WR Disables further send operations.

41391 SHUT_RDWR Disables further send and receive operations.

41392 The *shutdown()* function disables subsequent send and/or receive operations on a socket,
41393 depending on the value of the *how* argument.41394 **RETURN VALUE**41395 Upon successful completion, *shutdown()* shall return 0; otherwise, -1 shall be returned and *errno*
41396 set to indicate the error.41397 **ERRORS**41398 The *shutdown()* function shall fail if:41399 [EBADF] The *socket* argument is not a valid file descriptor.41400 [EINVAL] The *how* argument is invalid.

41401 [ENOTCONN] The socket is not connected.

41402 [ENOTSOCK] The *socket* argument does not refer to a socket.41403 The *shutdown()* function may fail if:

41404 [ENOBUFS] Insufficient resources were available in the system to perform the operation.

41405 **EXAMPLES**

41406 None.

41407 **APPLICATION USAGE**

41408 None.

41409 **RATIONALE**

41410 None.

41411 **FUTURE DIRECTIONS**

41412 None.

41413 **SEE ALSO**41414 *getsockopt()*, *read()*, *recv()*, *recvfrom()*, *recvmsg()*, *select()*, *send()*, *sendto()*, *setsockopt()*, *socket()*,
41415 *write()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>

41416 **CHANGE HISTORY**

41417 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

41418 NAME

41419 sigaction — examine and change a signal action

41420 SYNOPSIS

41421 CX

```
#include <signal.h>
```

```
41422 int sigaction(int sig, const struct sigaction *restrict act,
41423 struct sigaction *restrict oact);
41424
```

41425 DESCRIPTION

41426 The *sigaction()* function allows the calling process to examine and/or specify the action to be
 41427 associated with a specific signal. The argument *sig* specifies the signal; acceptable values are
 41428 defined in **<signal.h>**.

41429 The structure **sigaction**, used to describe an action to be taken, is defined in the **<signal.h>**
 41430 header to include at least the following members:

41431

Member Type	Member Name	Description
void(*) (int) sigset_t	<i>sa_handler</i> <i>sa_mask</i>	SIG_DFL, SIG_IGN, or pointer to a function. Additional set of signals to be blocked during execution of signal-catching function.
int void(*) (int, siginfo_t *, void *)	<i>sa_flags</i> <i>sa_sigaction</i>	Special flags to affect behavior of signal. Signal-catching function.

41440 The storage occupied by *sa_handler* and *sa_sigaction* may overlap, and a conforming application
 41441 shall not use both simultaneously.

41442 If the argument *act* is not a null pointer, it points to a structure specifying the action to be
 41443 associated with the specified signal. If the argument *oact* is not a null pointer, the action
 41444 previously associated with the signal is stored in the location pointed to by the argument *oact*. If
 41445 the argument *act* is a null pointer, signal handling is unchanged; thus, the call can be used to
 41446 enquire about the current handling of a given signal. The SIGKILL and SIGSTOP signals shall
 41447 not be added to the signal mask using this mechanism; this restriction shall be enforced by the
 41448 system without causing an error to be indicated.

41449 If the SA_SIGINFO flag (see below) is cleared in the *sa_flags* field of the **sigaction** structure, the
 41450 XSI|RTS *sa_handler* field identifies the action to be associated with the specified signal. If the
 41451 SA_SIGINFO flag is set in the *sa_flags* field, and the implementation supports the Realtime
 41452 Signals Extension option or the XSI Extension option, the *sa_sigaction* field specifies a signal-
 41453 catching function. If the SA_SIGINFO bit is cleared and the *sa_handler* field specifies a signal-
 41454 catching function, or if the SA_SIGINFO bit is set, the *sa_mask* field identifies a set of signals that
 41455 shall be added to the signal mask of the thread before the signal-catching function is invoked. If
 41456 the *sa_handler* field specifies a signal-catching function, the *sa_mask* field identifies a set of
 41457 signals that shall be added to the process' signal mask before the signal-catching function is
 41458 invoked.

41459 The *sa_flags* field can be used to modify the behavior of the specified signal.

41460 The following flags, defined in the **<signal.h>** header, can be set in *sa_flags*:

41461 XSI SA_NOCLDSTOP Do not generate SIGCHLD when children stop or stopped children
 41462 continue.

41463		If <i>sig</i> is SIGCHLD and the SA_NOCLDSTOP flag is not set in <i>sa_flags</i> , and
41464		the implementation supports the SIGCHLD signal, then a SIGCHLD
41465		signal shall be generated for the calling process whenever any of its child
41466 XSI		processes stop and a SIGCHLD signal may be generated for the calling
41467		process whenever any of its stopped child processes are continued. If <i>sig</i>
41468		is SIGCHLD and the SA_NOCLDSTOP flag is set in <i>sa_flags</i> , then the
41469		implementation shall not generate a SIGCHLD signal in this way.
41470 XSI	SA_ONSTACK	If set and an alternate signal stack has been declared with <i>sigaltstack()</i> , the
41471		signal shall be delivered to the calling process on that stack. Otherwise,
41472		the signal shall be delivered on the current stack.
41473 XSI	SA_RESETHAND	If set, the disposition of the signal shall be reset to SIG_DFL and the
41474		SA_SIGINFO flag shall be cleared on entry to the signal handler.
41475		Note: SIGILL and SIGTRAP cannot be automatically reset when delivered;
41476		the system silently enforces this restriction.
41477		Otherwise, the disposition of the signal shall not be modified on entry to
41478		the signal handler.
41479		In addition, if this flag is set, <i>sigaction()</i> behaves as if the SA_NODEFER
41480		flag were also set.
41481 XSI	SA_RESTART	This flag affects the behavior of interruptible functions; that is, those
41482		specified to fail with <i>errno</i> set to [EINTR]. If set, and a function specified
41483		as interruptible is interrupted by this signal, the function shall restart and
41484		shall not fail with [EINTR] unless otherwise specified. If the flag is not
41485		set, interruptible functions interrupted by this signal shall fail with <i>errno</i>
41486		set to [EINTR].
41487	SA_SIGINFO	If cleared and the signal is caught, the signal-catching function shall be
41488		entered as:
41489		<pre>void func(int signo);</pre>
41490		where <i>signo</i> is the only argument to the signal-catching function. In this
41491		case, the application shall use the <i>sa_handler</i> member to describe the
41492		signal-catching function and the application shall not modify the
41493		<i>sa_sigaction</i> member.
41494 XSI RTS		If SA_SIGINFO is set and the signal is caught, the signal-catching
41495		function shall be entered as:
41496		<pre>void func(int signo, siginfo_t *info, void *context);</pre>
41497		where two additional arguments are passed to the signal-catching
41498		function. The second argument shall point to an object of type siginfo_t
41499		explaining the reason why the signal was generated; the third argument
41500		can be cast to a pointer to an object of type ucontext_t to refer to the
41501		receiving process' context that was interrupted when the signal was
41502		delivered. In this case, the application shall use the <i>sa_sigaction</i> member
41503		to describe the signal-catching function and the application shall not
41504		modify the <i>sa_handler</i> member.
41505		The <i>si_signo</i> member contains the system-generated signal number.
41506 XSI		The <i>si_errno</i> member may contain implementation-defined additional
41507		error information; if non-zero, it contains an error number identifying the
41508		condition that caused the signal to be generated.

41509	XSI RTS	The <i>si_code</i> member contains a code identifying the cause of the signal.
41510	XSI	If the value of <i>si_code</i> is less than or equal to 0, then the signal was generated by a process and <i>si_pid</i> and <i>si_uid</i> , respectively, indicate the process ID and the real user ID of the sender. The <code><signal.h></code> header description contains information about the signal-specific contents of the elements of the siginfo_t type.
41511		
41512		
41513		
41514		
41515	XSI	If set, and <i>sig</i> equals SIGCHLD, child processes of the calling processes shall not be transformed into zombie processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited-for children that were transformed into zombie processes, it shall block until all of its children terminate, and <i>wait()</i> , <i>waitid()</i> , and <i>waitpid()</i> shall fail and set <i>errno</i> to [ECHILD]. Otherwise, terminating child processes shall be transformed into zombie processes, unless SIGCHLD is set to SIG_IGN.
41516		
41517		
41518		
41519		
41520		
41521		
41522		
41523	XSI	If set and <i>sig</i> is caught, <i>sig</i> shall not be added to the process' signal mask on entry to the signal handler unless it is included in <i>sa_mask</i> . Otherwise, <i>sig</i> shall always be added to the process' signal mask on entry to the signal handler.
41524		
41525		
41526		
41527		When a signal is caught by a signal-catching function installed by <i>sigaction()</i> , a new signal mask is calculated and installed for the duration of the signal-catching function (or until a call to either <i>sigprocmask()</i> or <i>sigsuspend()</i> is made). This mask is formed by taking the union of the current signal mask and the value of the <i>sa_mask</i> for the signal being delivered unless SA_NODEFER or SA_RESETHAND is set, and then including the signal being delivered. If and when the user's signal handler returns normally, the original signal mask is restored.
41528		
41529		
41530	XSI	
41531		
41532		Once an action is installed for a specific signal, it shall remain installed until another action is explicitly requested (by another call to <i>sigaction()</i>), until the SA_RESETHAND flag causes resetting of the handler, or until one of the <i>exec</i> functions is called.
41533		
41534	XSI	
41535		If the previous action for <i>sig</i> had been established by <i>signal()</i> , the values of the fields returned in the structure pointed to by <i>oact</i> are unspecified, and in particular <i>oact->sa_handler</i> is not necessarily the same value passed to <i>signal()</i> . However, if a pointer to the same structure or a copy thereof is passed to a subsequent call to <i>sigaction()</i> via the <i>act</i> argument, handling of the signal shall be as if the original call to <i>signal()</i> were repeated.
41536		
41537		
41538		
41539		
41540		If <i>sigaction()</i> fails, no new signal handler is installed.
41541		
41542		It is unspecified whether an attempt to set the action for a signal that cannot be caught or ignored to SIG_DFL is ignored or causes an error to be returned with <i>errno</i> set to [EINVAL].
41543		
41544		If SA_SIGINFO is not set in <i>sa_flags</i> , then the disposition of subsequent occurrences of <i>sig</i> when it is already pending is implementation-defined; the signal-catching function shall be invoked with a single argument. If the implementation supports the Realtime Signals Extension option, and if SA_SIGINFO is set in <i>sa_flags</i> , then subsequent occurrences of <i>sig</i> generated by <i>sigqueue()</i> or as a result of any signal-generating function that supports the specification of an application-defined value (when <i>sig</i> is already pending) shall be queued in FIFO order until delivered or accepted; the signal-catching function shall be invoked with three arguments. The application specified value is passed to the signal-catching function as the <i>si_value</i> member of the siginfo_t structure.
41545		
41546	RTS	
41547		
41548		
41549		
41550		
41551		
41552		The result of the use of <i>sigaction()</i> and a <i>sigwait()</i> function concurrently within a process on the same signal is unspecified.
41553		
41554		

41555 **RETURN VALUE**

41556 Upon successful completion, *sigaction()* shall return 0; otherwise, *-1* shall be returned, *errno* shall
 41557 be set to indicate the error, and no new signal-catching function shall be installed.

41558 **ERRORS**

41559 The *sigaction()* function shall fail if:

41560 [EINVAL] The *sig* argument is not a valid signal number or an attempt is made to catch a
 41561 signal that cannot be caught or ignore a signal that cannot be ignored.

41562 [ENOTSUP] The SA_SIGINFO bit flag is set in the *sa_flags* field of the **sigaction** structure,
 41563 and the implementation does not support either the Realtime Signals
 41564 Extension option, or the XSI Extension option.

41565 The *sigaction()* function may fail if:

41566 [EINVAL] An attempt was made to set the action to SIG_DFL for a signal that cannot be
 41567 caught or ignored (or both).

41568 **EXAMPLES**

41569 None.

41570 **APPLICATION USAGE**

41571 The *sigaction()* function supersedes the *signal()* function, and should be used in preference. In
 41572 particular, *sigaction()* and *signal()* should not be used in the same process to control the same
 41573 signal. The behavior of reentrant functions, as defined in the DESCRIPTION, is as specified by
 41574 this volume of IEEE Std 1003.1-2001, regardless of invocation from a signal-catching function.
 41575 This is the only intended meaning of the statement that reentrant functions may be used in
 41576 signal-catching functions without restrictions. Applications must still consider all effects of such
 41577 functions on such things as data structures, files, and process state. In particular, application
 41578 writers need to consider the restrictions on interactions when interrupting *sleep()* and
 41579 interactions among multiple handles for a file description. The fact that any specific function is
 41580 listed as reentrant does not necessarily mean that invocation of that function from a signal-
 41581 catching function is recommended.

41582 In order to prevent errors arising from interrupting non-reentrant function calls, applications
 41583 should protect calls to these functions either by blocking the appropriate signals or through the
 41584 use of some programmatic semaphore (see *semget()*, *sem_init()*, *sem_open()*, and so on). Note in
 41585 particular that even the “safe” functions may modify *errno*; the signal-catching function, if not
 41586 executing as an independent thread, may want to save and restore its value. Naturally, the same
 41587 principles apply to the reentrancy of application routines and asynchronous data access. Note
 41588 that *longjmp()* and *siglongjmp()* are not in the list of reentrant functions. This is because the code
 41589 executing after *longjmp()* and *siglongjmp()* can call any unsafe functions with the same danger as
 41590 calling those unsafe functions directly from the signal handler. Applications that use *longjmp()*
 41591 and *siglongjmp()* from within signal handlers require rigorous protection in order to be portable.
 41592 Many of the other functions that are excluded from the list are traditionally implemented using
 41593 either *malloc()* or *free()* functions or the standard I/O library, both of which traditionally use
 41594 data structures in a non-reentrant manner. Since any combination of different functions using a
 41595 common data structure can cause reentrancy problems, this volume of IEEE Std 1003.1-2001
 41596 does not define the behavior when any unsafe function is called in a signal handler that
 41597 interrupts an unsafe function.

41598 If the signal occurs other than as the result of calling *abort()*, *kill()*, or *raise()*, the behavior is
 41599 undefined if the signal handler calls any function in the standard library other than one of the
 41600 functions listed in the table above or refers to any object with static storage duration other than
 41601 by assigning a value to a static storage duration variable of type **volatile sig_atomic_t**.
 41602 Furthermore, if such a call fails, the value of *errno* is unspecified.

Usually, the signal is executed on the stack that was in effect before the signal was delivered. An alternate stack may be specified to receive a subset of the signals being caught.

When the signal handler returns, the receiving process resumes execution at the point it was interrupted unless the signal handler makes other arrangements. If *longjmp()* or *_longjmp()* is used to leave the signal handler, then the signal mask must be explicitly restored by the process.

This volume of IEEE Std 1003.1-2001 defines the third argument of a signal handling function when SA_SIGINFO is set as a **void *** instead of a **ucontext_t ***, but without requiring type checking. New applications should explicitly cast the third argument of the signal handling function to **ucontext_t ***.

The BSD optional four argument signal handling function is not supported by this volume of IEEE Std 1003.1-2001. The BSD declaration would be:

```
void handler(int sig, int code, struct sigcontext *scp,
             char *addr);
```

where *sig* is the signal number, *code* is additional information on certain signals, *scp* is a pointer to the **sigcontext** structure, and *addr* is additional address information. Much the same information is available in the objects pointed to by the second argument of the signal handler specified when SA_SIGINFO is set.

RATIONALE

Although this volume of IEEE Std 1003.1-2001 requires that signals that cannot be ignored shall not be added to the signal mask when a signal-catching function is entered, there is no explicit requirement that subsequent calls to *sigaction()* reflect this in the information returned in the *oact* argument. In other words, if SIGKILL is included in the *sa_mask* field of *act*, it is unspecified whether or not a subsequent call to *sigaction()* returns with SIGKILL included in the *sa_mask* field of *oact*.

The SA_NOCLDSTOP flag, when supplied in the *act->sa_flags* parameter, allows overloading SIGCHLD with the System V semantics that each SIGCLD signal indicates a single terminated child. Most conforming applications that catch SIGCHLD are expected to install signal-catching functions that repeatedly call the *waitpid()* function with the WNOHANG flag set, acting on each child for which status is returned, until *waitpid()* returns zero. If stopped children are not of interest, the use of the SA_NOCLDSTOP flag can prevent the overhead from invoking the signal-catching routine when they stop.

Some historical implementations also define other mechanisms for stopping processes, such as the *ptrace()* function. These implementations usually do not generate a SIGCHLD signal when processes stop due to this mechanism; however, that is beyond the scope of this volume of IEEE Std 1003.1-2001.

This volume of IEEE Std 1003.1-2001 requires that calls to *sigaction()* that supply a NULL *act* argument succeed, even in the case of signals that cannot be caught or ignored (that is, SIGKILL or SIGSTOP). The System V *signal()* and BSD *sigvec()* functions return [EINVAL] in these cases and, in this respect, their behavior varies from *sigaction()*.

This volume of IEEE Std 1003.1-2001 requires that *sigaction()* properly save and restore a signal action set up by the ISO C standard *signal()* function. However, there is no guarantee that the reverse is true, nor could there be given the greater amount of information conveyed by the **sigaction** structure. Because of this, applications should avoid using both functions for the same signal in the same process. Since this cannot always be avoided in case of general-purpose library routines, they should always be implemented with *sigaction()*.

It was intended that the *signal()* function should be implementable as a library routine using *sigaction()*.

41650 The POSIX Realtime Extension extends the *sigaction()* function as specified by the POSIX.1-1990
 41651 standard to allow the application to request on a per-signal basis via an additional signal action
 41652 flag that the extra parameters, including the application-defined signal value, if any, be passed
 41653 to the signal-catching function.

41654 FUTURE DIRECTIONS

41655 None.

41656 SEE ALSO

41657 Section 2.4 (on page 28), *bsd_signal()*, *kill()*, *_longjmp()*, *longjmp()*, *raise()*, *semget()*, *sem_init()*,
 41658 *sem_open()*, *sigaddset()*, *sigaltstack()*, *sigdelset()*, *sigemptyset()*, *sigfillset()*, *sigismember()*, *signal()*,
 41659 *sigprocmask()*, *sigsuspend()*, *wait()*, *waitid()*, *waitpid()*, the Base Definitions volume of
 41660 IEEE Std 1003.1-2001, <**signal.h**>, <**ucontext.h**>

41661 CHANGE HISTORY

41662 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

41663 Issue 5

41664 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and POSIX
 41665 Threads Extension.

41666 In the DESCRIPTION, the second argument to *func* when SA_SIGINFO is set is no longer
 41667 permitted to be NULL, and the description of permitted **siginfo_t** contents is expanded by
 41668 reference to <**signal.h**>.

41669 Since the X/OPEN UNIX Extension functionality is now folded into the BASE, the [ENOTSUP]
 41670 error is deleted.

41671 Issue 6

41672 The Open Group Corrigendum U028/7 is applied. In the paragraph entitled “Signal Effects on
 41673 Other Functions”, a reference to *sigpending()* is added.

41674 In the DESCRIPTION, the text “Signal Generation and Delivery”, “Signal Actions”, and “Signal
 41675 Effects on Other Functions” are moved to a separate section of this volume of
 41676 IEEE Std 1003.1-2001.

41677 Text describing functionality from the Realtime Signals option is marked.

41678 The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- 41679 • The [ENOTSUP] error condition is added.

41680 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

41681 The **restrict** keyword is added to the *sigaction()* prototype for alignment with the
 41682 ISO/IEC 9899: 1999 standard.

41683 References to the *wait3()* function are removed.

41684 The SYNOPSIS is marked CX since the presence of this function in the <**signal.h**> header is an
 41685 extension over the ISO C standard.

41686 **NAME**

41687 sigaddset — add a signal to a signal set

41688 **SYNOPSIS**

41689 CX #include <signal.h>

41690 int sigaddset(sigset_t *set, int signo);

41691

41692 **DESCRIPTION**41693 The *sigaddset()* function adds the individual signal specified by the *signo* to the signal set pointed
41694 to by *set*.41695 Applications shall call either *sigemptyset()* or *sigfillset()* at least once for each object of type
41696 **sigset_t** prior to any other use of that object. If such an object is not initialized in this way, but is
41697 nonetheless supplied as an argument to any of *pthread_sigmask()*, *sigaction()*, *sigaddset()*,
41698 *sigdelset()*, *sigismember()*, *sigpending()*, *sigprocmask()*, *sigsuspend()*, *sigtimedwait()*, *sigwait()*, or
41699 *sigwaitinfo()*, the results are undefined.41700 **RETURN VALUE**41701 Upon successful completion, *sigaddset()* shall return 0; otherwise, it shall return -1 and set *errno*
41702 to indicate the error.41703 **ERRORS**41704 The *sigaddset()* function may fail if:41705 [EINVAL] The value of the *signo* argument is an invalid or unsupported signal number.41706 **EXAMPLES**

41707 None.

41708 **APPLICATION USAGE**

41709 None.

41710 **RATIONALE**

41711 None.

41712 **FUTURE DIRECTIONS**

41713 None.

41714 **SEE ALSO**41715 Section 2.4 (on page 28), *sigaction()*, *sigdelset()*, *sigemptyset()*, *sigfillset()*, *sigismember()*,
41716 *sigpending()*, *sigprocmask()*, *sigsuspend()*, the Base Definitions volume of IEEE Std 1003.1-2001,
41717 <signal.h>41718 **CHANGE HISTORY**

41719 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

41720 **Issue 5**41721 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in
41722 previous issues.41723 **Issue 6**

41724 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

41725 The SYNOPSIS is marked CX since the presence of this function in the <signal.h> header is an
41726 extension over the ISO C standard.

41727 **NAME**

41728 sigaltstack — set and get signal alternate stack context

41729 **SYNOPSIS**41730 XSI

```
#include <signal.h>
```

41731

```
int sigaltstack(const stack_t *restrict ss, stack_t *restrict oss);
```

41732

41733 **DESCRIPTION**

41734 The *sigaltstack()* function allows a process to define and examine the state of an alternate stack
 41735 for signal handlers. Signals that have been explicitly declared to execute on the alternate stack
 41736 shall be delivered on the alternate stack.

41737 If *ss* is not a null pointer, it points to a **stack_t** structure that specifies the alternate signal stack
 41738 that shall take effect upon return from *sigaltstack()*. The *ss_flags* member specifies the new stack
 41739 state. If it is set to *SS_DISABLE*, the stack is disabled and *ss_sp* and *ss_size* are ignored.
 41740 Otherwise, the stack shall be enabled, and the *ss_sp* and *ss_size* members specify the new address
 41741 and size of the stack.

41742 The range of addresses starting at *ss_sp* up to but not including *ss_sp+ss_size* is available to the
 41743 implementation for use as the stack. This function makes no assumptions regarding which end
 41744 is the stack base and in which direction the stack grows as items are pushed.

41745 If *oss* is not a null pointer, on successful completion it shall point to a **stack_t** structure that
 41746 specifies the alternate signal stack that was in effect prior to the call to *sigaltstack()*. The *ss_sp*
 41747 and *ss_size* members specify the address and size of that stack. The *ss_flags* member specifies the
 41748 stack's state, and may contain one of the following values:

41749 **SS_ONSTACK** The process is currently executing on the alternate signal stack. Attempts to
 41750 modify the alternate signal stack while the process is executing on it fail. This
 41751 flag shall not be modified by processes.

41752 **SS_DISABLE** The alternate signal stack is currently disabled.

41753 The value *SIGSTKSZ* is a system default specifying the number of bytes that would be used to
 41754 cover the usual case when manually allocating an alternate stack area. The value *MINSIGSTKSZ*
 41755 is defined to be the minimum stack size for a signal handler. In computing an alternate stack
 41756 size, a program should add that amount to its stack requirements to allow for the system
 41757 implementation overhead. The constants *SS_ONSTACK*, *SS_DISABLE*, *SIGSTKSZ*, and
 41758 *MINSIGSTKSZ* are defined in **<signal.h>**.

41759 After a successful call to one of the *exec* functions, there are no alternate signal stacks in the new
 41760 process image.

41761 In some implementations, a signal (whether or not indicated to execute on the alternate stack)
 41762 shall always execute on the alternate stack if it is delivered while another signal is being caught
 41763 using the alternate stack.

41764 Use of this function by library threads that are not bound to kernel-scheduled entities results in
 41765 undefined behavior.

41766 **RETURN VALUE**

41767 Upon successful completion, *sigaltstack()* shall return 0; otherwise, it shall return *-1* and set *errno*
 41768 to indicate the error.

41769 **ERRORS**41770 The *sigaltstack()* function shall fail if:

41771 [EINVAL] The *ss* argument is not a null pointer, and the *ss_flags* member pointed to by *ss*
 41772 contains flags other than *SS_DISABLE*.

41773 [ENOMEM] The size of the alternate stack area is less than *MINSIGSTKSZ*.

41774 [EPERM] An attempt was made to modify an active stack.

41775 **EXAMPLES**41776 **Allocating Memory for an Alternate Stack**

41777 The following example illustrates a method for allocating memory for an alternate stack.

```
41778 #include <signal.h>
41779 ...
41780 if ((sigstk.ss_sp = malloc(SIGSTKSZ)) == NULL)
41781     /* Error return. */
41782     sigstk.ss_size = SIGSTKSZ;
41783     sigstk.ss_flags = 0;
41784     if (sigaltstack(&sigstk, (stack_t *)0) < 0)
41785         perror("sigaltstack");
```

41786 **APPLICATION USAGE**

41787 On some implementations, stack space is automatically extended as needed. On those
 41788 implementations, automatic extension is typically not available for an alternate stack. If the stack
 41789 overflows, the behavior is undefined.

41790 **RATIONALE**

41791 None.

41792 **FUTURE DIRECTIONS**

41793 None.

41794 **SEE ALSO**

41795 Section 2.4 (on page 28), *sigaction()*, *sigsetjmp()*, the Base Definitions volume of
 41796 IEEE Std 1003.1-2001, <signal.h>

41797 **CHANGE HISTORY**

41798 First released in Issue 4, Version 2.

41799 **Issue 5**

41800 Moved from X/OPEN UNIX extension to BASE.

41801 The last sentence of the DESCRIPTION was included as an APPLICATION USAGE note in
 41802 previous issues.

41803 **Issue 6**

41804 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

41805 The **restrict** keyword is added to the *sigaltstack()* prototype for alignment with the
 41806 ISO/IEC 9899:1999 standard.

41807 **NAME**

41808 sigdelset — delete a signal from a signal set

41809 **SYNOPSIS**41810 CX `#include <signal.h>`41811 `int sigdelset(sigset_t *set, int signo);`

41812

41813 **DESCRIPTION**41814 The *sigdelset()* function deletes the individual signal specified by *signo* from the signal set
41815 pointed to by *set*.41816 Applications should call either *sigemptyset()* or *sigfillset()* at least once for each object of type
41817 **sigset_t** prior to any other use of that object. If such an object is not initialized in this way, but is
41818 nonetheless supplied as an argument to any of *pthread_sigmask()*, *sigaction()*, *sigaddset()*,
41819 *sigdelset()*, *sigismember()*, *sigpending()*, *sigprocmask()*, *sigsuspend()*, *sigtimedwait()*, *sigwait()*, or
41820 *sigwaitinfo()*, the results are undefined.41821 **RETURN VALUE**41822 Upon successful completion, *sigdelset()* shall return 0; otherwise, it shall return -1 and set *errno*
41823 to indicate the error.41824 **ERRORS**41825 The *sigdelset()* function may fail if:41826 [EINVAL] The *signo* argument is not a valid signal number, or is an unsupported signal
41827 number.41828 **EXAMPLES**

41829 None.

41830 **APPLICATION USAGE**

41831 None.

41832 **RATIONALE**

41833 None.

41834 **FUTURE DIRECTIONS**

41835 None.

41836 **SEE ALSO**41837 Section 2.4 (on page 28), *sigaction()*, *sigaddset()*, *sigemptyset()*, *sigfillset()*, *sigismember()*,
41838 *sigpending()*, *sigprocmask()*, *sigsuspend()*, the Base Definitions volume of IEEE Std 1003.1-2001,
41839 **<signal.h>**41840 **CHANGE HISTORY**

41841 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

41842 **Issue 5**41843 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in
41844 previous issues.41845 **Issue 6**41846 The SYNOPSIS is marked CX since the presence of this function in the **<signal.h>** header is an
41847 extension over the ISO C standard.

41848 **NAME**

41849 sigemptyset — initialize and empty a signal set

41850 **SYNOPSIS**41851 CX `#include <signal.h>`41852 `int sigemptyset(sigset_t *set);`

41853

41854 **DESCRIPTION**41855 The *sigemptyset()* function initializes the signal set pointed to by *set*, such that all signals defined
41856 in IEEE Std 1003.1-2001 are excluded.41857 **RETURN VALUE**41858 Upon successful completion, *sigemptyset()* shall return 0; otherwise, it shall return *-1* and set
41859 *errno* to indicate the error.41860 **ERRORS**

41861 No errors are defined.

41862 **EXAMPLES**

41863 None.

41864 **APPLICATION USAGE**

41865 None.

41866 **RATIONALE**41867 The implementation of the *sigemptyset()* (or *sigfillset()*) function could quite trivially clear (or
41868 set) all the bits in the signal set. Alternatively, it would be reasonable to initialize part of the
41869 structure, such as a version field, to permit binary-compatibility between releases where the size
41870 of the set varies. For such reasons, either *sigemptyset()* or *sigfillset()* must be called prior to any
41871 other use of the signal set, even if such use is read-only (for example, as an argument to
41872 *sigpending()*). This function is not intended for dynamic allocation.41873 The *sigfillset()* and *sigemptyset()* functions require that the resulting signal set include (or
41874 exclude) all the signals defined in this volume of IEEE Std 1003.1-2001. Although it is outside the
41875 scope of this volume of IEEE Std 1003.1-2001 to place this requirement on signals that are
41876 implemented as extensions, it is recommended that implementation-defined signals also be
41877 affected by these functions. However, there may be a good reason for a particular signal not to
41878 be affected. For example, blocking or ignoring an implementation-defined signal may have
41879 undesirable side effects, whereas the default action for that signal is harmless. In such a case, it
41880 would be preferable for such a signal to be excluded from the signal set returned by *sigfillset()*.41881 In early proposals there was no distinction between invalid and unsupported signals (the names
41882 of optional signals that were not supported by an implementation were not defined by that
41883 implementation). The [EINVAL] error was thus specified as a required error for invalid signals.
41884 With that distinction, it is not necessary to require implementations of these functions to
41885 determine whether an optional signal is actually supported, as that could have a significant
41886 performance impact for little value. The error could have been required for invalid signals and
41887 optional for unsupported signals, but this seemed unnecessarily complex. Thus, the error is
41888 optional in both cases.41889 **FUTURE DIRECTIONS**

41890 None.

41891 **SEE ALSO**

41892 Section 2.4 (on page 28), *sigaction()*, *sigaddset()*, *sigdelset()*, *sigfillset()*, *sigismember()*, *sigpending()*,
41893 *sigprocmask()*, *sigsuspend()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**signal.h**>

41894 **CHANGE HISTORY**

41895 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

41896 **Issue 6**

41897 The SYNOPSIS is marked CX since the presence of this function in the <**signal.h**> header is an
41898 extension over the ISO C standard.

41899 **NAME**

41900 sigfillset — initialize and fill a signal set

41901 **SYNOPSIS**

41902 CX #include <signal.h>

41903 int sigfillset(sigset_t *set);

41904

41905 **DESCRIPTION**41906 The *sigfillset()* function shall initialize the signal set pointed to by *set*, such that all signals
41907 defined in this volume of IEEE Std 1003.1-2001 are included.41908 **RETURN VALUE**41909 Upon successful completion, *sigfillset()* shall return 0; otherwise, it shall return -1 and set *errno*
41910 to indicate the error.41911 **ERRORS**

41912 No errors are defined.

41913 **EXAMPLES**

41914 None.

41915 **APPLICATION USAGE**

41916 None.

41917 **RATIONALE**41918 Refer to *sigemptyset()* (on page 1348).41919 **FUTURE DIRECTIONS**

41920 None.

41921 **SEE ALSO**41922 Section 2.4 (on page 28), *sigaction()*, *sigaddset()*, *sigdelset()*, *sigemptyset()*, *sigismember()*,
41923 *sigpending()*, *sigprocmask()*, *sigsuspend()*, the Base Definitions volume of IEEE Std 1003.1-2001,
41924 <signal.h>41925 **CHANGE HISTORY**

41926 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

41927 **Issue 6**41928 The SYNOPSIS is marked CX since the presence of this function in the <signal.h> header is an
41929 extension over the ISO C standard.

41930 **NAME**

41931 sighold, sigignore, sigpause, sigrelse, sigset — signal management

41932 **SYNOPSIS**

```

41933 XSI      #include <signal.h>

41934          int sighold(int sig);
41935          int sigignore(int sig);
41936          int sigpause(int sig);
41937          int sigrelse(int sig);
41938          void (*sigset(int sig, void (*disp)(int)))(int);
41939

```

41940 **DESCRIPTION**

41941 Use of any of these functions is unspecified in a multi-threaded process.

41942 The *sighold()*, *sigignore()*, *sigpause()*, *sigrelse()*, and *sigset()* functions provide simplified signal management.

41943

41944 The *sigset()* function shall modify signal dispositions. The *sig* argument specifies the signal, which may be any signal except SIGKILL and SIGSTOP. The *disp* argument specifies the signal's disposition, which may be SIG_DFL, SIG_IGN, or the address of a signal handler. If *sigset()* is used, and *disp* is the address of a signal handler, the system shall add *sig* to the calling process' signal mask before executing the signal handler; when the signal handler returns, the system shall restore the calling process' signal mask to its state prior to the delivery of the signal. In addition, if *sigset()* is used, and *disp* is equal to SIG_HOLD, *sig* shall be added to the calling process' signal mask and *sig*'s disposition shall remain unchanged. If *sigset()* is used, and *disp* is not equal to SIG_HOLD, *sig* shall be removed from the calling process' signal mask.

41953 The *sighold()* function shall add *sig* to the calling process' signal mask.41954 The *sigrelse()* function shall remove *sig* from the calling process' signal mask.41955 The *sigignore()* function shall set the disposition of *sig* to SIG_IGN.

41956 The *sigpause()* function shall remove *sig* from the calling process' signal mask and suspend the calling process until a signal is received. The *sigpause()* function shall restore the process' signal mask to its original state before returning.

41959 If the action for the SIGCHLD signal is set to SIG_IGN, child processes of the calling processes shall not be transformed into zombie processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited-for children that were transformed into zombie processes, it shall block until all of its children terminate, and *wait()*, *waitid()*, and *waitpid()* shall fail and set *errno* to [ECHILD].

41964 **RETURN VALUE**

41965 Upon successful completion, *sigset()* shall return SIG_HOLD if the signal had been blocked and the signal's previous disposition if it had not been blocked. Otherwise, SIG_ERR shall be returned and *errno* set to indicate the error.

41968 The *sigpause()* function shall suspend execution of the thread until a signal is received, whereupon it shall return -1 and set *errno* to [EINTR].

41970 For all other functions, upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to indicate the error.

41971

41972 **ERRORS**

41973 These functions shall fail if:

41974 [EINVAL] The *sig* argument is an illegal signal number.41975 The *sigset()* and *sigignore()* functions shall fail if:41976 [EINVAL] An attempt is made to catch a signal that cannot be caught, or to ignore a
41977 signal that cannot be ignored.41978 **EXAMPLES**

41979 None.

41980 **APPLICATION USAGE**41981 The *sigaction()* function provides a more comprehensive and reliable mechanism for controlling
41982 signals; new applications should use *sigaction()* rather than *sigset()*.41983 The *sighold()* function, in conjunction with *sigelse()* or *sigpause()*, may be used to establish
41984 critical regions of code that require the delivery of a signal to be temporarily deferred.41985 The *sigsuspend()* function should be used in preference to *sigpause()* for broader portability.41986 **RATIONALE**

41987 None.

41988 **FUTURE DIRECTIONS**

41989 None.

41990 **SEE ALSO**41991 Section 2.4 (on page 28), *exec*, *pause()*, *sigaction()*, *signal()*, *sigsuspend()*, *waitid()*, the Base
41992 Definitions volume of IEEE Std 1003.1-2001, <signal.h>41993 **CHANGE HISTORY**

41994 First released in Issue 4, Version 2.

41995 **Issue 5**

41996 Moved from X/OPEN UNIX extension to BASE.

41997 The DESCRIPTION is updated to indicate that the *sigpause()* function restores the process'
41998 signal mask to its original state before returning.41999 The RETURN VALUE section is updated to indicate that the *sigpause()* function suspends
42000 execution of the process until a signal is received, whereupon it returns -1 and sets *errno* to
42001 [EINTR].42002 **Issue 6**

42003 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

42004 References to the *wait3()* function are removed.

42005 The XSI functions are split out into their own reference page.

42006 NAME

42007 siginterrupt — allow signals to interrupt functions

42008 SYNOPSIS

42009 XSI #include <signal.h>

42010 int siginterrupt(int *sig*, int *flag*);

42011

42012 DESCRIPTION

42013 The *siginterrupt()* function shall change the restart behavior when a function is interrupted by
42014 the specified signal. The function *siginterrupt(sig, flag)* has an effect as if implemented as:

```
42015       siginterrupt(int sig, int flag) {
42016           int ret;
42017           struct sigaction act;

42018           (void) sigaction(sig, NULL, &act);
42019           if (flag)
42020               act.sa_flags &= ~SA_RESTART;
42021           else
42022               act.sa_flags |= SA_RESTART;
42023           ret = sigaction(sig, &act, NULL);
42024           return ret;
42025       }
```

42026 RETURN VALUE

42027 Upon successful completion, *siginterrupt()* shall return 0; otherwise, -1 shall be returned and
42028 *errno* set to indicate the error.

42029 ERRORS

42030 The *siginterrupt()* function shall fail if:

42031 [EINVAL] The *sig* argument is not a valid signal number.

42032 EXAMPLES

42033 None.

42034 APPLICATION USAGE

42035 The *siginterrupt()* function supports programs written to historical system interfaces. A
42036 conforming application, when being written or rewritten, should use *sigaction()* with the
42037 SA_RESTART flag instead of *siginterrupt()*.

42038 RATIONALE

42039 None.

42040 FUTURE DIRECTIONS

42041 None.

42042 SEE ALSO

42043 Section 2.4 (on page 28), *sigaction()*, the Base Definitions volume of IEEE Std 1003.1-2001,
42044 <signal.h>

42045 CHANGE HISTORY

42046 First released in Issue 4, Version 2.

42047 **Issue 5**

42048 Moved from X/OPEN UNIX extension to BASE.

42049 **NAME**

42050 sigismember — test for a signal in a signal set

42051 **SYNOPSIS**42052 CX `#include <signal.h>`42053 `int sigismember(const sigset_t *set, int signo);`

42054

42055 **DESCRIPTION**42056 The *sigismember()* function shall test whether the signal specified by *signo* is a member of the set
42057 pointed to by *set*.42058 Applications should call either *sigemptyset()* or *sigfillset()* at least once for each object of type
42059 **sigset_t** prior to any other use of that object. If such an object is not initialized in this way, but is
42060 nonetheless supplied as an argument to any of *pthread_sigmask()*, *sigaction()*, *sigaddset()*,
42061 *sigdelset()*, *sigismember()*, *sigpending()*, *sigprocmask()*, *sigsuspend()*, *sigtimedwait()*, *sigwait()*, or
42062 *sigwaitinfo()*, the results are undefined.42063 **RETURN VALUE**42064 Upon successful completion, *sigismember()* shall return 1 if the specified signal is a member of
42065 the specified set, or 0 if it is not. Otherwise, it shall return -1 and set *errno* to indicate the error.42066 **ERRORS**42067 The *sigismember()* function may fail if:42068 [EINVAL] The *signo* argument is not a valid signal number, or is an unsupported signal
42069 number.42070 **EXAMPLES**

42071 None.

42072 **APPLICATION USAGE**

42073 None.

42074 **RATIONALE**

42075 None.

42076 **FUTURE DIRECTIONS**

42077 None.

42078 **SEE ALSO**42079 Section 2.4 (on page 28), *sigaction()*, *sigaddset()*, *sigdelset()*, *sigfillset()*, *sigemptyset()*, *sigpending()*,
42080 *sigprocmask()*, *sigsuspend()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**signal.h**>42081 **CHANGE HISTORY**

42082 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

42083 **Issue 5**42084 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in
42085 previous issues.42086 **Issue 6**42087 The SYNOPSIS is marked CX since the presence of this function in the <**signal.h**> header is an
42088 extension over the ISO C standard.

42089 **NAME**

42090 siglongjmp — non-local goto with signal handling

42091 **SYNOPSIS**

42092 CX #include <setjmp.h>

42093 void siglongjmp(sigjmp_buf env, int val);

42094

42095 **DESCRIPTION**42096 The *siglongjmp()* function shall be equivalent to the *longjmp()* function, except as follows:

- 42097 • References to *setjmp()* shall be equivalent to *sigsetjmp()*.
- 42098 • The *siglongjmp()* function shall restore the saved signal mask if and only if the *env* argument
- 42099 was initialized by a call to *sigsetjmp()* with a non-zero *savemask* argument.

42100 **RETURN VALUE**

42101 After *siglongjmp()* is completed, program execution shall continue as if the corresponding

42102 invocation of *sigsetjmp()* had just returned the value specified by *val*. The *siglongjmp()* function

42103 shall not cause *sigsetjmp()* to return 0; if *val* is 0, *sigsetjmp()* shall return the value 1.

42104 **ERRORS**

42105 No errors are defined.

42106 **EXAMPLES**

42107 None.

42108 **APPLICATION USAGE**

42109 The distinction between *setjmp()* or *longjmp()* and *sigsetjmp()* or *siglongjmp()* is only significant

42110 for programs which use *sigaction()*, *sigprocmask()*, or *sigsuspend()*.

42111 **RATIONALE**

42112 None.

42113 **FUTURE DIRECTIONS**

42114 None.

42115 **SEE ALSO**

42116 *longjmp()*, *setjmp()*, *sigprocmask()*, *sigsetjmp()*, *sigsuspend()*, the Base Definitions volume of

42117 IEEE Std 1003.1-2001, <setjmp.h>

42118 **CHANGE HISTORY**

42119 First released in Issue 3. Included for alignment with the ISO POSIX-1 standard.

42120 **Issue 5**

42121 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

42122 **Issue 6**42123 The DESCRIPTION is rewritten in terms of *longjmp()*.

42124 The SYNOPSIS is marked CX since the presence of this function in the <setjmp.h> header is an

42125 extension over the ISO C standard.

42126 **NAME**

42127 signal — signal management

42128 **SYNOPSIS**

42129 #include <signal.h>

42130 void (*signal(int *sig*, void (**func*)(int)))(int);42131 **DESCRIPTION**

42132 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 42133 conflict between the requirements described here and the ISO C standard is unintentional. This
 42134 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

42135 CX Use of this function is unspecified in a multi-threaded process.

42136 The *signal()* function chooses one of three ways in which receipt of the signal number *sig* is to be
 42137 subsequently handled. If the value of *func* is SIG_DFL, default handling for that signal shall
 42138 occur. If the value of *func* is SIG_IGN, the signal shall be ignored. Otherwise, the application
 42139 shall ensure that *func* points to a function to be called when that signal occurs. An invocation of
 42140 such a function because of a signal, or (recursively) of any further functions called by that
 42141 invocation (other than functions in the standard library), is called a “signal handler”.

42142 When a signal occurs, and *func* points to a function, it is implementation-defined whether the
 42143 equivalent of a:

42144 signal(*sig*, SIG_DFL);

42145 is executed or the implementation prevents some implementation-defined set of signals (at least
 42146 including *sig*) from occurring until the current signal handling has completed. (If the value of *sig*
 42147 is SIGILL, the implementation may alternatively define that no action is taken.) Next the
 42148 equivalent of:

42149 (*func)(*sig*);

42150 is executed. If and when the function returns, if the value of *sig* was SIGFPE, SIGILL, or
 42151 SIGSEGV or any other implementation-defined value corresponding to a computational
 42152 exception, the behavior is undefined. Otherwise, the program shall resume execution at the
 42153 CX point it was interrupted. If the signal occurs as the result of calling the *abort()*, *raise()*, *kill()*,
 42154 *pthread_kill()*, or *sigqueue()* function, the signal handler shall not call the *raise()* function.

42155 CX If the signal occurs other than as the result of calling *abort()*, *raise()*, *kill()*, *pthread_kill()*, or
 42156 *sigqueue()*, the behavior is undefined if the signal handler refers to any object with static storage
 42157 duration other than by assigning a value to an object declared as volatile **sig_atomic_t**, or if the
 42158 signal handler calls any function in the standard library other than one of the functions listed in
 42159 Section 2.4 (on page 28). Furthermore, if such a call fails, the value of *errno* is unspecified.

42160 At program start-up, the equivalent of:

42161 signal(*sig*, SIG_IGN);

42162 is executed for some signals, and the equivalent of:

42163 signal(*sig*, SIG_DFL);

42164 CX is executed for all other signals (see *exec*).

42165 **RETURN VALUE**

42166 If the request can be honored, *signal()* shall return the value of *func* for the most recent call to
 42167 *signal()* for the specified signal *sig*. Otherwise, SIG_ERR shall be returned and a positive value
 42168 shall be stored in *errno*.

42169 ERRORS

42170 The *signal()* function shall fail if:

42171 CX [EINVAL] The *sig* argument is not a valid signal number or an attempt is made to catch a
 42172 signal that cannot be caught or ignore a signal that cannot be ignored.

42173 The *signal()* function may fail if:

42174 CX [EINVAL] An attempt was made to set the action to SIG_DFL for a signal that cannot be
 42175 caught or ignored (or both).

42176 EXAMPLES

42177 None.

42178 APPLICATION USAGE

42179 The *sigaction()* function provides a more comprehensive and reliable mechanism for controlling
 42180 signals; new applications should use *sigaction()* rather than *signal()*.

42181 RATIONALE

42182 None.

42183 FUTURE DIRECTIONS

42184 None.

42185 SEE ALSO

42186 Section 2.4 (on page 28), *exec*, *pause()*, *sigaction()*, *sigsuspend()*, *waitid()*, the Base Definitions
 42187 volume of IEEE Std 1003.1-2001, <**signal.h**>

42188 CHANGE HISTORY

42189 First released in Issue 1. Derived from Issue 1 of the SVID.

42190 Issue 5

42191 Moved from X/OPEN UNIX extension to BASE.

42192 The DESCRIPTION is updated to indicate that the *sigpause()* function restores the process'
 42193 signal mask to its original state before returning.

42194 The RETURN VALUE section is updated to indicate that the *sigpause()* function suspends
 42195 execution of the process until a signal is received, whereupon it returns -1 and sets *errno* to
 42196 [EINTR].

42197 Issue 6

42198 Extensions beyond the ISO C standard are marked.

42199 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

42200 The DESCRIPTION is updated for alignment with the ISO/IEC 9899:1999 standard.

42201 References to the *wait3()* function are removed.

42202 The *sighold()*, *sigignore()*, *sigelse()*, and *sigset()* functions are split out onto their own reference
 42203 page.

42204 **NAME**

42205 signbit — test sign

42206 **SYNOPSIS**

42207 #include <math.h>

42208 int signbit(real-floating x);

42209 **DESCRIPTION**

42210 cx The functionality described on this reference page is aligned with the ISO C standard. Any
42211 conflict between the requirements described here and the ISO C standard is unintentional. This
42212 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

42213 The *signbit()* macro shall determine whether the sign of its argument value is negative. NaNs,
42214 zeros, and infinities have a sign bit.

42215 **RETURN VALUE**

42216 The *signbit()* macro shall return a non-zero value if and only if the sign of its argument value is
42217 negative.

42218 **ERRORS**

42219 No errors are defined.

42220 **EXAMPLES**

42221 None.

42222 **APPLICATION USAGE**

42223 None.

42224 **RATIONALE**

42225 None.

42226 **FUTURE DIRECTIONS**

42227 None.

42228 **SEE ALSO**

42229 *fpclassify()*, *isfinite()*, *isinf()*, *isnan()*, *isnormal()*, the Base Definitions volume of
42230 IEEE Std 1003.1-2001, <math.h>

42231 **CHANGE HISTORY**

42232 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

42233 NAME

42234 sigpause — remove a signal from the signal mask and suspend the thread

42235 SYNOPSIS

42236 XSI `#include <signal.h>`

42237 `int sigpause(int sig);`

42238

42239 DESCRIPTION

42240 Refer to *sighold()*.

42241 **NAME**

42242 sigpending — examine pending signals

42243 **SYNOPSIS**42244 CX `#include <signal.h>`42245 `int sigpending(sigset_t *set);`

42246

42247 **DESCRIPTION**

42248 The *sigpending()* function shall store, in the location referenced by the *set* argument, the set of
 42249 signals that are blocked from delivery to the calling thread and that are pending on the process
 42250 or the calling thread.

42251 **RETURN VALUE**

42252 Upon successful completion, *sigpending()* shall return 0; otherwise, -1 shall be returned and
 42253 *errno* set to indicate the error.

42254 **ERRORS**

42255 No errors are defined.

42256 **EXAMPLES**

42257 None.

42258 **APPLICATION USAGE**

42259 None.

42260 **RATIONALE**

42261 None.

42262 **FUTURE DIRECTIONS**

42263 None.

42264 **SEE ALSO**

42265 *sigaddset()*, *sigdelset()*, *sigemptyset()*, *sigfillset()*, *sigismember()*, *sigprocmask()*, the Base Definitions
 42266 volume of IEEE Std 1003.1-2001, <**signal.h**>

42267 **CHANGE HISTORY**

42268 First released in Issue 3.

42269 **Issue 5**

42270 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

42271 **Issue 6**

42272 The SYNOPSIS is marked CX since the presence of this function in the <**signal.h**> header is an
 42273 extension over the ISO C standard.

42274 NAME

42275 sigprocmask — examine and change blocked signals

42276 SYNOPSIS

42277 CX `#include <signal.h>`

42278 `int sigprocmask(int how, const sigset_t *restrict set,`
42279 `sigset_t *restrict oset);`

42280

42281 DESCRIPTION

42282 Refer to *pthread_sigmask()*.

42283 **NAME**42284 sigqueue — queue a signal to a process (**REALTIME**)42285 **SYNOPSIS**

42286 RTS #include <signal.h>

42287 int sigqueue(pid_t pid, int signo, const union sigval value);

42288

42289 **DESCRIPTION**

42290 The *sigqueue()* function shall cause the signal specified by *signo* to be sent with the value
 42291 specified by *value* to the process specified by *pid*. If *signo* is zero (the null signal), error checking
 42292 is performed but no signal is actually sent. The null signal can be used to check the validity of
 42293 *pid*.

42294 The conditions required for a process to have permission to queue a signal to another process
 42295 are the same as for the *kill()* function.

42296 The *sigqueue()* function shall return immediately. If SA_SIGINFO is set for *signo* and if the
 42297 resources were available to queue the signal, the signal shall be queued and sent to the receiving
 42298 process. If SA_SIGINFO is not set for *signo*, then *signo* shall be sent at least once to the receiving
 42299 process; it is unspecified whether *value* shall be sent to the receiving process as a result of this
 42300 call.

42301 If the value of *pid* causes *signo* to be generated for the sending process, and if *signo* is not blocked
 42302 for the calling thread and if no other thread has *signo* unblocked or is waiting in a *sigwait()*
 42303 function for *signo*, either *signo* or at least the pending, unblocked signal shall be delivered to the
 42304 calling thread before the *sigqueue()* function returns. Should any multiple pending signals in the
 42305 range SIGRTMIN to SIGRTMAX be selected for delivery, it shall be the lowest numbered one.
 42306 The selection order between realtime and non-realtime signals, or between multiple pending
 42307 non-realtime signals, is unspecified.

42308 **RETURN VALUE**

42309 Upon successful completion, the specified signal shall have been queued, and the *sigqueue()*
 42310 function shall return a value of zero. Otherwise, the function shall return a value of -1 and set
 42311 *errno* to indicate the error.

42312 **ERRORS**42313 The *sigqueue()* function shall fail if:

42314 [EAGAIN] No resources are available to queue the signal. The process has already
 42315 queued {SIGQUEUE_MAX} signals that are still pending at the receiver(s), or
 42316 a system-wide resource limit has been exceeded.

42317 [EINVAL] The value of the *signo* argument is an invalid or unsupported signal number.

42318 [EPERM] The process does not have the appropriate privilege to send the signal to the
 42319 receiving process.

42320 [ESRCH] The process *pid* does not exist.

42321 **EXAMPLES**

42322 None.

42323 **APPLICATION USAGE**

42324 None.

42325 **RATIONALE**

42326 The *sigqueue()* function allows an application to queue a realtime signal to itself or to another
42327 process, specifying the application-defined value. This is common practice in realtime
42328 applications on existing realtime systems. It was felt that specifying another function in the
42329 *sig...* name space already carved out for signals was preferable to extending the interface to
42330 *kill()*.

42331 Such a function became necessary when the put/get event function of the message queues was
42332 removed. It should be noted that the *sigqueue()* function implies reduced performance in a
42333 security-conscious implementation as the access permissions between the sender and receiver
42334 have to be checked on each send when the *pid* is resolved into a target process. Such access
42335 checks were necessary only at message queue open in the previous interface.

42336 The standard developers required that *sigqueue()* have the same semantics with respect to the
42337 null signal as *kill()*, and that the same permission checking be used. But because of the difficulty
42338 of implementing the “broadcast” semantic of *kill()* (for example, to process groups) and the
42339 interaction with resource allocation, this semantic was not adopted. The *sigqueue()* function
42340 queues a signal to a single process specified by the *pid* argument.

42341 The *sigqueue()* function can fail if the system has insufficient resources to queue the signal. An
42342 explicit limit on the number of queued signals that a process could send was introduced. While
42343 the limit is “per-sender”, this volume of IEEE Std 1003.1-2001 does not specify that the resources
42344 be part of the state of the sender. This would require either that the sender be maintained after
42345 exit until all signals that it had sent to other processes were handled or that all such signals that
42346 had not yet been acted upon be removed from the queue(s) of the receivers. This volume of
42347 IEEE Std 1003.1-2001 does not preclude this behavior, but an implementation that allocated
42348 queuing resources from a system-wide pool (with per-sender limits) and that leaves queued
42349 signals pending after the sender exits is also permitted.

42350 **FUTURE DIRECTIONS**

42351 None.

42352 **SEE ALSO**

42353 Section 2.8.1 (on page 41), the Base Definitions volume of IEEE Std 1003.1-2001, <signal.h>

42354 **CHANGE HISTORY**

42355 First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the
42356 POSIX Threads Extension.

42357 **Issue 6**42358 The *sigqueue()* function is marked as part of the Realtime Signals Extension option.

42359 The [ENOSYS] error condition has been removed as stubs need not be provided if an
42360 implementation does not support the Realtime Signals Extension option.

42361 **NAME**

42362 sigrelse, sigset — signal management

42363 **SYNOPSIS**42364 XSI `#include <signal.h>`42365 `int sigrelse(int sig);`42366 `void (*sigset(int sig, void (*disp)(int)))(int);`

42367

42368 **DESCRIPTION**42369 Refer to *sighold()*.

42370 **NAME**

42371 sigsetjmp — set jump point for a non-local goto

42372 **SYNOPSIS**42373 CX `#include <setjmp.h>`42374 `int sigsetjmp(sigjmp_buf env, int savemask);`

42375

42376 **DESCRIPTION**42377 The *sigsetjmp()* function shall be equivalent to the *setjmp()* function, except as follows:

- 42378 • References to *setjmp()* are equivalent to *sigsetjmp()*.
- 42379 • References to *longjmp()* are equivalent to *siglongjmp()*.
- 42380 • If the value of the *savemask* argument is not 0, *sigsetjmp()* shall also save the current signal mask of the calling thread as part of the calling environment.

42382 **RETURN VALUE**

42383 If the return is from a successful direct invocation, *sigsetjmp()* shall return 0. If the return is from
 42384 a call to *siglongjmp()*, *sigsetjmp()* shall return a non-zero value.

42385 **ERRORS**

42386 No errors are defined.

42387 **EXAMPLES**

42388 None.

42389 **APPLICATION USAGE**

42390 The distinction between *setjmp()/longjmp()* and *sigsetjmp()/siglongjmp()* is only significant for
 42391 programs which use *sigaction()*, *sigprocmask()*, or *sigsuspend()*.

42392 Note that since this function is defined in terms of *setjmp()*, if *savemask* is zero, it is unspecified
 42393 whether the signal mask is saved.

42394 **RATIONALE**

42395 The ISO C standard specifies various restrictions on the usage of the *setjmp()* macro in order to
 42396 permit implementors to recognize the name in the compiler and not implement an actual
 42397 function. These same restrictions apply to the *sigsetjmp()* macro.

42398 There are processors that cannot easily support these calls, but this was not considered a
 42399 sufficient reason to exclude them.

42400 4.2 BSD, 4.3 BSD, and XSI-conformant systems provide functions named *_setjmp()* and
 42401 *_longjmp()* that, together with *setjmp()* and *longjmp()*, provide the same functionality as
 42402 *sigsetjmp()* and *siglongjmp()*. On those systems, *setjmp()* and *longjmp()* save and restore signal
 42403 masks, while *_setjmp()* and *_longjmp()* do not. On System V Release 3 and in corresponding
 42404 issues of the SVID, *setjmp()* and *longjmp()* are explicitly defined not to save and restore signal
 42405 masks. In order to permit existing practice in both cases, the relation of *setjmp()* and *longjmp()* to
 42406 signal masks is not specified, and a new set of functions is defined instead.

42407 The *longjmp()* and *siglongjmp()* functions operate as in the previous issue provided the matching
 42408 *setjmp()* or *sigsetjmp()* has been performed in the same thread. Non-local jumps into contexts
 42409 saved by other threads would be at best a questionable practice and were not considered worthy
 42410 of standardization.

42411 **FUTURE DIRECTIONS**

42412 None.

42413 **SEE ALSO**

42414 *siglongjmp()*, *signal()*, *sigprocmask()*, *sigsuspend()*, the Base Definitions volume of
42415 IEEE Std 1003.1-2001, <**setjmp.h**>

42416 **CHANGE HISTORY**

42417 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

42418 **Issue 5**

42419 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

42420 **Issue 6**42421 The DESCRIPTION is reworded in terms of *setjmp()*.

42422 The SYNOPSIS is marked CX since the presence of this function in the <**setjmp.h**> header is an
42423 extension over the ISO C standard.

42424 **NAME**

42425 sigsuspend — wait for a signal

42426 **SYNOPSIS**42427 CX

```
#include <signal.h>
```

42428

```
int sigsuspend(const sigset_t *sigmask);
```

42429

42430 **DESCRIPTION**

42431 The *sigsuspend()* function shall replace the current signal mask of the calling thread with the set
 42432 of signals pointed to by *sigmask* and then suspend the thread until delivery of a signal whose
 42433 action is either to execute a signal-catching function or to terminate the process. This shall not
 42434 cause any other signals that may have been pending on the process to become pending on the
 42435 thread.

42436 If the action is to terminate the process then *sigsuspend()* shall never return. If the action is to
 42437 execute a signal-catching function, then *sigsuspend()* shall return after the signal-catching
 42438 function returns, with the signal mask restored to the set that existed prior to the *sigsuspend()*
 42439 call.

42440 It is not possible to block signals that cannot be ignored. This is enforced by the system without
 42441 causing an error to be indicated.

42442 **RETURN VALUE**

42443 Since *sigsuspend()* suspends thread execution indefinitely, there is no successful completion
 42444 return value. If a return occurs, *-1* shall be returned and *errno* set to indicate the error.

42445 **ERRORS**42446 The *sigsuspend()* function shall fail if:

42447 [EINTR] A signal is caught by the calling process and control is returned from the
 42448 signal-catching function.

42449 **EXAMPLES**

42450 None.

42451 **APPLICATION USAGE**

42452 Normally, at the beginning of a critical code section, a specified set of signals is blocked using
 42453 the *sigprocmask()* function. When the thread has completed the critical section and needs to wait
 42454 for the previously blocked signal(s), it pauses by calling *sigsuspend()* with the mask that was
 42455 returned by the *sigprocmask()* call.

42456 **RATIONALE**

42457 None.

42458 **FUTURE DIRECTIONS**

42459 None.

42460 **SEE ALSO**

42461 Section 2.4 (on page 28), *pause()*, *sigaction()*, *sigaddset()*, *sigdelset()*, *sigemptyset()*, *sigfillset()*, the
 42462 Base Definitions volume of IEEE Std 1003.1-2001, *<signal.h>*

42463 **CHANGE HISTORY**

42464 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

42465 **Issue 5**

42466 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

42467 **Issue 6**

42468 The text in the RETURN VALUE section has been changed from “suspends process execution”
42469 to “suspends thread execution”. This reflects IEEE PASC Interpretation 1003.1c #40.

42470 Text in the APPLICATION USAGE section has been replaced.

42471 The SYNOPSIS is marked CX since the presence of this function in the <signal.h> header is an
42472 extension over the ISO C standard.

42473 NAME

42474 sigtimedwait, sigwaitinfo — wait for queued signals (**REALTIME**)

42475 SYNOPSIS

42476 RTS

```
#include <signal.h>
```

```

42477 int sigtimedwait(const sigset_t *restrict set,
42478                 siginfo_t *restrict info,
42479                 const struct timespec *restrict timeout);
42480 int sigwaitinfo(const sigset_t *restrict set,
42481                 siginfo_t *restrict info);
42482

```

42483 DESCRIPTION

42484 The *sigtimedwait()* function shall be equivalent to *sigwaitinfo()* except that if none of the signals
 42485 specified by *set* are pending, *sigtimedwait()* shall wait for the time interval specified in the
 42486 **timespec** structure referenced by *timeout*. If the **timespec** structure pointed to by *timeout* is
 42487 zero-valued and if none of the signals specified by *set* are pending, then *sigtimedwait()* shall
 42488 MON return immediately with an error. If *timeout* is the NULL pointer, the behavior is unspecified. If
 42489 the Monotonic Clock option is supported, the **CLOCK_MONOTONIC** clock shall be used to
 42490 measure the time interval specified by the *timeout* argument.

42491 The *sigwaitinfo()* function selects the pending signal from the set specified by *set*. Should any of
 42492 multiple pending signals in the range SIGRTMIN to SIGRTMAX be selected, it shall be the
 42493 lowest numbered one. The selection order between realtime and non-realtime signals, or
 42494 between multiple pending non-realtime signals, is unspecified. If no signal in *set* is pending at
 42495 the time of the call, the calling thread shall be suspended until one or more signals in *set* become
 42496 pending or until it is interrupted by an unblocked, caught signal.

42497 The *sigwaitinfo()* function shall be equivalent to the *sigwait()* function if the *info* argument is
 42498 NULL. If the *info* argument is non-NULL, the *sigwaitinfo()* function shall be equivalent to
 42499 *sigwait()*, except that the selected signal number shall be stored in the *si_signo* member, and the
 42500 cause of the signal shall be stored in the *si_code* member. If any value is queued to the selected
 42501 signal, the first such queued value shall be dequeued and, if the *info* argument is non-NULL, the
 42502 value shall be stored in the *si_value* member of *info*. The system resource used to queue the
 42503 signal shall be released and returned to the system for other use. If no value is queued, the
 42504 content of the *si_value* member is undefined. If no further signals are queued for the selected
 42505 signal, the pending indication for that signal shall be reset.

42506 RETURN VALUE

42507 Upon successful completion (that is, one of the signals specified by *set* is pending or is
 42508 generated) *sigwaitinfo()* and *sigtimedwait()* shall return the selected signal number. Otherwise,
 42509 the function shall return a value of -1 and set *errno* to indicate the error.

42510 ERRORS

42511 The *sigtimedwait()* function shall fail if:42512 [EAGAIN] No signal specified by *set* was generated within the specified timeout period.42513 The *sigtimedwait()* and *sigwaitinfo()* functions may fail if:

42514 [EINTR] The wait was interrupted by an unblocked, caught signal. It shall be
 42515 documented in system documentation whether this error causes these
 42516 functions to fail.

42517 The *sigtimedwait()* function may also fail if:

42518 [EINVAL] The *timeout* argument specified a *tv_nsec* value less than zero or greater than
 42519 or equal to 1 000 million.

42520 An implementation only checks for this error if no signal is pending in *set* and it is necessary to
 42521 wait.

42522 EXAMPLES

42523 None.

42524 APPLICATION USAGE

42525 The *sigtimedwait()* function times out and returns an [EAGAIN] error. Application writers
 42526 should note that this is inconsistent with other functions such as *pthread_cond_timedwait()* that
 42527 return [ETIMEDOUT].

42528 RATIONALE

42529 Existing programming practice on realtime systems uses the ability to pause waiting for a
 42530 selected set of events and handle the first event that occurs in-line instead of in a signal-handling
 42531 function. This allows applications to be written in an event-directed style similar to a state
 42532 machine. This style of programming is useful for largescale transaction processing in which the
 42533 overall throughput of an application and the ability to clearly track states are more important
 42534 than the ability to minimize the response time of individual event handling.

42535 It is possible to construct a signal-waiting macro function out of the realtime signal function
 42536 mechanism defined in this volume of IEEE Std 1003.1-2001. However, such a macro has to
 42537 include the definition of a generalized handler for all signals to be waited on. A significant
 42538 portion of the overhead of handler processing can be avoided if the signal-waiting function is
 42539 provided by the kernel. This volume of IEEE Std 1003.1-2001 therefore provides two signal-
 42540 waiting functions—one that waits indefinitely and one with a timeout—as part of the overall
 42541 realtime signal function specification.

42542 The specification of a function with a timeout allows an application to be written that can be
 42543 broken out of a wait after a set period of time if no event has occurred. It was argued that setting
 42544 a timer event before the wait and recognizing the timer event in the wait would also implement
 42545 the same functionality, but at a lower performance level. Because of the performance
 42546 degradation associated with the user-level specification of a timer event and the subsequent
 42547 cancelation of that timer event after the wait completes for a valid event, and the complexity
 42548 associated with handling potential race conditions associated with the user-level method, the
 42549 separate function has been included.

42550 Note that the semantics of the *sigwaitinfo()* function are nearly identical to that of the *sigwait()*
 42551 function defined by this volume of IEEE Std 1003.1-2001. The only difference is that *sigwaitinfo()*
 42552 returns the queued signal value in the *value* argument. The return of the queued value is
 42553 required so that applications can differentiate between multiple events queued to the same
 42554 signal number.

42555 The two distinct functions are being maintained because some implementations may choose to
 42556 implement the POSIX Threads Extension functions and not implement the queued signals
 42557 extensions. Note, though, that *sigwaitinfo()* does not return the queued value if the *value*
 42558 argument is NULL, so the POSIX Threads Extension *sigwait()* function can be implemented as a
 42559 macro on *sigwaitinfo()*.

42560 The *sigtimedwait()* function was separated from the *sigwaitinfo()* function to address concerns
 42561 regarding the overloading of the *timeout* pointer to indicate indefinite wait (no timeout), timed
 42562 wait, and immediate return, and concerns regarding consistency with other functions where the
 42563 conditional and timed waits were separate functions from the pure blocking function. The
 42564 semantics of *sigtimedwait()* are specified such that *sigwaitinfo()* could be implemented as a
 42565 macro with a NULL pointer for *timeout*.

42566 The *sigwait* functions provide a synchronous mechanism for threads to wait for
 42567 asynchronously-generated signals. One important question was how many threads that are
 42568 suspended in a call to a *sigwait*() function for a signal should return from the call when the
 42569 signal is sent. Four choices were considered:

- 42570 1. Return an error for multiple simultaneous calls to *sigwait* functions for the same signal.
- 42571 2. One or more threads return.
- 42572 3. All waiting threads return.
- 42573 4. Exactly one thread returns.

42574 Prohibiting multiple calls to *sigwait*() for the same signal was felt to be overly restrictive. The
 42575 “one or more” behavior made implementation of conforming packages easy at the expense of
 42576 forcing POSIX threads clients to protect against multiple simultaneous calls to *sigwait*() in
 42577 application code in order to achieve predictable behavior. There was concern that the “all
 42578 waiting threads” behavior would result in “signal broadcast storms”, consuming excessive CPU
 42579 resources by replicating the signals in the general case. Furthermore, no convincing examples
 42580 could be presented that delivery to all was either simpler or more powerful than delivery to one.

42581 Thus, the consensus was that exactly one thread that was suspended in a call to a *sigwait*
 42582 function for a signal should return when that signal occurs. This is not an onerous restriction as:

- 42583 • A multi-way signal wait can be built from the single-way wait.
- 42584 • Signals should only be handled by application-level code, as library routines cannot guess
 42585 what the application wants to do with signals generated for the entire process.
- 42586 • Applications can thus arrange for a single thread to wait for any given signal and call any
 42587 needed routines upon its arrival.

42588 In an application that is using signals for interprocess communication, signal processing is
 42589 typically done in one place. Alternatively, if the signal is being caught so that process cleanup
 42590 can be done, the signal handler thread can call separate process cleanup routines for each
 42591 portion of the application. Since the application main line started each portion of the application,
 42592 it is at the right abstraction level to tell each portion of the application to clean up.

42593 Certainly, there exist programming styles where it is logical to consider waiting for a single
 42594 signal in multiple threads. A simple *sigwait_multiple*() routine can be constructed to achieve this
 42595 goal. A possible implementation would be to have each *sigwait_multiple*() caller registered as
 42596 having expressed interest in a set of signals. The caller then waits on a thread-specific condition
 42597 variable. A single server thread calls a *sigwait*() function on the union of all registered signals.
 42598 When the *sigwait*() function returns, the appropriate state is set and condition variables are
 42599 broadcast. New *sigwait_multiple*() callers may cause the pending *sigwait*() call to be canceled
 42600 and reissued in order to update the set of signals being waited for.

42601 FUTURE DIRECTIONS

42602 None.

42603 SEE ALSO

42604 Section 2.8.1 (on page 41), *pause*(), *pthread_sigmask*(), *sigaction*(), *sigpending*(), *sigsuspend*(),
 42605 *sigwait*(), the Base Definitions volume of IEEE Std 1003.1-2001, <**signal.h**>, <**time.h**>

42606 CHANGE HISTORY

42607 First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the
 42608 POSIX Threads Extension.

42609 **Issue 6**

- 42610 These functions are marked as part of the Realtime Signals Extension option.
- 42611 The Open Group Corrigendum U035/3 is applied. The SYNOPSIS of the *sigwaitinfo()* function
42612 has been corrected so that the second argument is of type **siginfo_t** *.
- 42613 The [ENOSYS] error condition has been removed as stubs need not be provided if an
42614 implementation does not support the Realtime Signals Extension option.
- 42615 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that the
42616 CLOCK_MONOTONIC clock, if supported, is used to measure timeout intervals.
- 42617 The **restrict** keyword is added to the *sigtimedwait()* and *sigwaitinfo()* prototypes for alignment
42618 with the ISO/IEC 9899:1999 standard.

42619 **NAME**

42620 sigwait — wait for queued signals

42621 **SYNOPSIS**42622 CX `#include <signal.h>`42623 `int sigwait(const sigset_t *restrict set, int *restrict sig);`

42624

42625 **DESCRIPTION**

42626 The *sigwait()* function shall select a pending signal from *set*, atomically clear it from the system's
 42627 set of pending signals, and return that signal number in the location referenced by *sig*. If prior to
 42628 the call to *sigwait()* there are multiple pending instances of a single signal number, it is
 42629 implementation-defined whether upon successful return there are any remaining pending
 42630 signals for that signal number. If the implementation supports queued signals and there are
 42631 multiple signals queued for the signal number selected, the first such queued signal shall cause a
 42632 return from *sigwait()* and the remainder shall remain queued. If no signal in *set* is pending at the
 42633 time of the call, the thread shall be suspended until one or more becomes pending. The signals
 42634 defined by *set* shall have been blocked at the time of the call to *sigwait()*; otherwise, the behavior
 42635 is undefined. The effect of *sigwait()* on the signal actions for the signals in *set* is unspecified.

42636 If more than one thread is using *sigwait()* to wait for the same signal, no more than one of these
 42637 threads shall return from *sigwait()* with the signal number. Which thread returns from *sigwait()*
 42638 if more than a single thread is waiting is unspecified.

42639 RTS Should any of the multiple pending signals in the range SIGRTMIN to SIGRTMAX be selected, it
 42640 shall be the lowest numbered one. The selection order between realtime and non-realtime
 42641 signals, or between multiple pending non-realtime signals, is unspecified.

42642 **RETURN VALUE**

42643 Upon successful completion, *sigwait()* shall store the signal number of the received signal at the
 42644 location referenced by *sig* and return zero. Otherwise, an error number shall be returned to
 42645 indicate the error.

42646 **ERRORS**42647 The *sigwait()* function may fail if:

42648 [EINVAL] The *set* argument contains an invalid or unsupported signal number.

42649 **EXAMPLES**

42650 None.

42651 **APPLICATION USAGE**

42652 None.

42653 **RATIONALE**

42654 To provide a convenient way for a thread to wait for a signal, this volume of
 42655 IEEE Std 1003.1-2001 provides the *sigwait()* function. For most cases where a thread has to wait
 42656 for a signal, the *sigwait()* function should be quite convenient, efficient, and adequate.

42657 However, requests were made for a lower-level primitive than *sigwait()* and for semaphores that
 42658 could be used by threads. After some consideration, threads were allowed to use semaphores
 42659 and *sem_post()* was defined to be async-signal and async-cancel-safe.

42660 In summary, when it is necessary for code run in response to an asynchronous signal to notify a
 42661 thread, *sigwait()* should be used to handle the signal. Alternatively, if the implementation
 42662 provides semaphores, they also can be used, either following *sigwait()* or from within a signal
 42663 handling routine previously registered with *sigaction()*.

42664 **FUTURE DIRECTIONS**

42665 None.

42666 **SEE ALSO**

42667 Section 2.4 (on page 28), Section 2.8.1 (on page 41), *pause()*, *pthread_sigmask()*, *sigaction()*,
42668 *sigpending()*, *sigsuspend()*, *sigwaitinfo()*, the Base Definitions volume of IEEE Std 1003.1-2001,
42669 <**signal.h**>, <**time.h**>

42670 **CHANGE HISTORY**

42671 First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the
42672 POSIX Threads Extension.

42673 **Issue 6**

42674 The **restrict** keyword is added to the *sigwait()* prototype for alignment with the
42675 ISO/IEC 9899:1999 standard.

42676 NAME

42677 sigwaitinfo — wait for queued signals (**REALTIME**)

42678 SYNOPSIS

42679 RTS #include <signal.h>

42680 int sigwaitinfo(const sigset_t *restrict set, siginfo_t *restrict info);

42681

42682 DESCRIPTION

42683 Refer to *sigtimedwait()*.

42684 **NAME**

42685 sin, sinf, sinl — sine function

42686 **SYNOPSIS**

42687 #include <math.h>

42688 double sin(double x);

42689 float sinf(float x);

42690 long double sinl(long double x);

42691 **DESCRIPTION**

42692 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 42693 conflict between the requirements described here and the ISO C standard is unintentional. This
 42694 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

42695 These functions shall compute the sine of their argument *x*, measured in radians.

42696 An application wishing to check for error situations should set *errno* to zero and call
 42697 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 42698 *fetetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 42699 zero, an error has occurred.

42700 **RETURN VALUE**42701 Upon successful completion, these functions shall return the sine of *x*.42702 **MX** If *x* is NaN, a NaN shall be returned.42703 If *x* is ± 0 , *x* shall be returned.42704 If *x* is subnormal, a range error may occur and *x* should be returned.

42705 If *x* is $\pm \text{Inf}$, a domain error shall occur, and either a NaN (if supported), or an implementation-
 42706 defined value shall be returned.

42707 **ERRORS**

42708 These functions shall fail if:

42709 **MX** **Domain Error** The *x* argument is $\pm \text{Inf}$.

42710 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 42711 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling
 42712 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 42713 shall be raised.

42714 These functions may fail if:

42715 **MX** **Range Error** The value of *x* is subnormal

42716 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 42717 then *errno* shall be set to [ERANGE]. If the integer expression
 42718 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
 42719 floating-point exception shall be raised.

42720 **EXAMPLES**42721 **Taking the Sine of a 45-Degree Angle**

```
42722     #include <math.h>
42723     ...
42724     double radians = 45.0 * M_PI / 180;
42725     double result;
42726     ...
42727     result = sin(radians);
```

42728 **APPLICATION USAGE**

42729 These functions may lose accuracy when their argument is near a multiple of π or is far from 0.0.

42730 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
42731 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

42732 **RATIONALE**

42733 None.

42734 **FUTURE DIRECTIONS**

42735 None.

42736 **SEE ALSO**

42737 *asin()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-2001,
42738 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

42739 **CHANGE HISTORY**

42740 First released in Issue 1. Derived from Issue 1 of the SVID.

42741 **Issue 5**

42742 The last two paragraphs of the DESCRIPTION were included as APPLICATION USAGE notes
42743 in previous issues.

42744 **Issue 6**

42745 The *sinf()* and *sinl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

42746 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
42747 revised to align with the ISO/IEC 9899:1999 standard.

42748 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
42749 marked.

42750 **NAME**

42751 sinh, sinh, sinhl — hyperbolic sine functions

42752 **SYNOPSIS**

42753 #include <math.h>

42754 double sinh(double x);

42755 float sinh(float x);

42756 long double sinhl(long double x);

42757 **DESCRIPTION**

42758 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 42759 conflict between the requirements described here and the ISO C standard is unintentional. This
 42760 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

42761 These functions shall compute the hyperbolic sine of their argument *x*.

42762 An application wishing to check for error situations should set *errno* to zero and call
 42763 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 42764 *fetetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 42765 zero, an error has occurred.

42766 **RETURN VALUE**42767 Upon successful completion, these functions shall return the hyperbolic sine of *x*.

42768 If the result would cause an overflow, a range error shall occur and $\pm\text{HUGE_VAL}$,
 42769 $\pm\text{HUGE_VALF}$, and $\pm\text{HUGE_VALL}$ (with the same sign as *x*) shall be returned as appropriate for
 42770 the type of the function.

42771 MX If *x* is NaN, a NaN shall be returned.42772 If *x* is ± 0 or $\pm\text{Inf}$, *x* shall be returned.42773 If *x* is subnormal, a range error may occur and *x* should be returned.42774 **ERRORS**

42775 These functions shall fail if:

42776 Range Error The result would cause an overflow.

42777 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 42778 then *errno* shall be set to [ERANGE]. If the integer expression
 42779 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow
 42780 floating-point exception shall be raised.

42781 These functions may fail if:

42782 MX Range Error The value *x* is subnormal.

42783 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 42784 then *errno* shall be set to [ERANGE]. If the integer expression
 42785 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
 42786 floating-point exception shall be raised.

42787 **EXAMPLES**

42788 None.

42789 **APPLICATION USAGE**

42790 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
42791 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

42792 **RATIONALE**

42793 None.

42794 **FUTURE DIRECTIONS**

42795 None.

42796 **SEE ALSO**

42797 *asinh()*, *cosh()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, *tanh()*, the Base Definitions volume of
42798 IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
42799 <math.h>

42800 **CHANGE HISTORY**

42801 First released in Issue 1. Derived from Issue 1 of the SVID.

42802 **Issue 5**

42803 The DESCRIPTION is updated to indicate how an application should check for an error. This
42804 text was previously published in the APPLICATION USAGE section.

42805 **Issue 6**42806 The *sinhf()* and *sinhl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

42807 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
42808 revised to align with the ISO/IEC 9899:1999 standard.

42809 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
42810 marked.

42811 **NAME**

42812 sinl — sine function

42813 **SYNOPSIS**

42814 #include <math.h>

42815 long double sinl(long double x);

42816 **DESCRIPTION**42817 Refer to *sin()*.

42818 **NAME**

42819 sleep — suspend execution for an interval of time

42820 **SYNOPSIS**

42821 #include <unistd.h>

42822 unsigned sleep(unsigned *seconds*);42823 **DESCRIPTION**

42824 The *sleep()* function shall cause the calling thread to be suspended from execution until either
 42825 the number of realtime seconds specified by the argument *seconds* has elapsed or a signal is
 42826 delivered to the calling thread and its action is to invoke a signal-catching function or to
 42827 terminate the process. The suspension time may be longer than requested due to the scheduling
 42828 of other activity by the system.

42829 If a SIGALRM signal is generated for the calling process during execution of *sleep()* and if the
 42830 SIGALRM signal is being ignored or blocked from delivery, it is unspecified whether *sleep()*
 42831 returns when the SIGALRM signal is scheduled. If the signal is being blocked, it is also
 42832 unspecified whether it remains pending after *sleep()* returns or it is discarded.

42833 If a SIGALRM signal is generated for the calling process during execution of *sleep()*, except as a
 42834 result of a prior call to *alarm()*, and if the SIGALRM signal is not being ignored or blocked from
 42835 delivery, it is unspecified whether that signal has any effect other than causing *sleep()* to return.

42836 If a signal-catching function interrupts *sleep()* and examines or changes either the time a
 42837 SIGALRM is scheduled to be generated, the action associated with the SIGALRM signal, or
 42838 whether the SIGALRM signal is blocked from delivery, the results are unspecified.

42839 If a signal-catching function interrupts *sleep()* and calls *siglongjmp()* or *longjmp()* to restore an
 42840 environment saved prior to the *sleep()* call, the action associated with the SIGALRM signal and
 42841 the time at which a SIGALRM signal is scheduled to be generated are unspecified. It is also
 42842 unspecified whether the SIGALRM signal is blocked, unless the process' signal mask is restored
 42843 as part of the environment.

42844 XSI Interactions between *sleep()* and any of *setitimer()*, *ualarm()*, or *usleep()* are unspecified.

42845 **RETURN VALUE**

42846 If *sleep()* returns because the requested time has elapsed, the value returned shall be 0. If *sleep()*
 42847 returns due to delivery of a signal, the return value shall be the “unslept” amount (the requested
 42848 time minus the time actually slept) in seconds.

42849 **ERRORS**

42850 No errors are defined.

42851 **EXAMPLES**

42852 None.

42853 **APPLICATION USAGE**

42854 None.

42855 **RATIONALE**

42856 There are two general approaches to the implementation of the *sleep()* function. One is to use the
 42857 *alarm()* function to schedule a SIGALRM signal and then suspend the process waiting for that
 42858 signal. The other is to implement an independent facility. This volume of IEEE Std 1003.1-2001
 42859 permits either approach.

42860 In order to comply with the requirement that no primitive shall change a process attribute unless
 42861 explicitly described by this volume of IEEE Std 1003.1-2001, an implementation using SIGALRM
 42862 must carefully take into account any SIGALRM signal scheduled by previous *alarm()* calls, the

action previously established for SIGALRM, and whether SIGALRM was blocked. If a SIGALRM has been scheduled before the *sleep()* would ordinarily complete, the *sleep()* must be shortened to that time and a SIGALRM generated (possibly simulated by direct invocation of the signal-catching function) before *sleep()* returns. If a SIGALRM has been scheduled after the *sleep()* would ordinarily complete, it must be rescheduled for the same time before *sleep()* returns. The action and blocking for SIGALRM must be saved and restored.

Historical implementations often implement the SIGALRM-based version using *alarm()* and *pause()*. One such implementation is prone to infinite hangups, as described in *pause()*. Another such implementation uses the C-language *setjmp()* and *longjmp()* functions to avoid that window. That implementation introduces a different problem: when the SIGALRM signal interrupts a signal-catching function installed by the user to catch a different signal, the *longjmp()* aborts that signal-catching function. An implementation based on *sigprocmask()*, *alarm()*, and *sigsuspend()* can avoid these problems.

Despite all reasonable care, there are several very subtle, but detectable and unavoidable, differences between the two types of implementations. These are the cases mentioned in this volume of IEEE Std 1003.1-2001 where some other activity relating to SIGALRM takes place, and the results are stated to be unspecified. All of these cases are sufficiently unusual as not to be of concern to most applications.

See also the discussion of the term *realtime* in *alarm()*.

Since *sleep()* can be implemented using *alarm()*, the discussion about alarms occurring early under *alarm()* applies to *sleep()* as well.

Application writers should note that the type of the argument *seconds* and the return value of *sleep()* is **unsigned**. That means that a Strictly Conforming POSIX System Interfaces Application cannot pass a value greater than the minimum guaranteed value for {UINT_MAX}, which the ISO C standard sets as 65 535, and any application passing a larger value is restricting its portability. A different type was considered, but historical implementations, including those with a 16-bit **int** type, consistently use either **unsigned** or **int**.

Scheduling delays may cause the process to return from the *sleep()* function significantly after the requested time. In such cases, the return value should be set to zero, since the formula (requested time minus the time actually spent) yields a negative number and *sleep()* returns an **unsigned**.

FUTURE DIRECTIONS

None.

SEE ALSO

alarm(), *getitimer()*, *nanosleep()*, *pause()*, *sigaction()*, *sigsetjmp()*, *ualarm()*, *usleep()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**unistd.h**>

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

42903 NAME

42904 snprintf — print formatted output

42905 SYNOPSIS

42906 #include <stdio.h>

42907 int snprintf(char *restrict *s*, size_t *n*,

42908 const char *restrict *format*, ...);

42909 DESCRIPTION

42910 Refer to *fprintf()*.

42911 **NAME**

42912 socketmark — determine whether a socket is at the out-of-band mark

42913 **SYNOPSIS**

42914 #include <sys/socket.h>

42915 int socketmark(int s);

42916 **DESCRIPTION**

42917 The *socketmark()* function shall determine whether the socket specified by the descriptor *s* is at
 42918 the out-of-band data mark (see the System Interfaces volume of IEEE Std 1003.1-2001, Section
 42919 2.10.12, Socket Out-of-Band Data State). If the protocol for the socket supports out-of-band data
 42920 by marking the stream with an out-of-band data mark, the *socketmark()* function shall return 1
 42921 when all data preceding the mark has been read and the out-of-band data mark is the first
 42922 element in the receive queue. The *socketmark()* function shall not remove the mark from the
 42923 stream.

42924 **RETURN VALUE**

42925 Upon successful completion, the *socketmark()* function shall return a value indicating whether
 42926 the socket is at an out-of-band data mark. If the protocol has marked the data stream and all data
 42927 preceding the mark has been read, the return value shall be 1; if there is no mark, or if data
 42928 precedes the mark in the receive queue, the *socketmark()* function shall return 0. Otherwise, it
 42929 shall return a value of -1 and set *errno* to indicate the error.

42930 **ERRORS**42931 The *socketmark()* function shall fail if:42932 [EBADF] The *s* argument is not a valid file descriptor.42933 [ENOTTY] The *s* argument does not specify a descriptor for a socket.42934 **EXAMPLES**

42935 None.

42936 **APPLICATION USAGE**

42937 The use of this function between receive operations allows an application to determine which
 42938 received data precedes the out-of-band data and which follows the out-of-band data.

42939 There is an inherent race condition in the use of this function. On an empty receive queue, the
 42940 current read of the location might well be at the “mark”, but the system has no way of knowing
 42941 that the next data segment that will arrive from the network will carry the mark, and
 42942 *socketmark()* will return false, and the next read operation will silently consume the mark.

42943 Hence, this function can only be used reliably when the application already knows that the out-
 42944 of-band data has been seen by the system or that it is known that there is data waiting to be read
 42945 at the socket (via SIGURG or *select()*). See Section 2.10.11 (on page 61), Section 2.10.12 (on page
 42946 61), Section 2.10.14 (on page 62), and *pselect()* for details.

42947 **RATIONALE**

42948 The *socketmark()* function replaces the historical SIOCATMARK command to *ioctl()* which
 42949 implemented the same functionality on many implementations. Using a wrapper function
 42950 follows the adopted conventions to avoid specifying commands to the *ioctl()* function, other
 42951 than those now included to support XSI STREAMS. The *socketmark()* function could be
 42952 implemented as follows:

42953 #include <sys/ioctl.h>

42954 int socketmark(int s)

42955 {


```
42956         int val;
42957         if (ioctl(s,SIOCATMARK,&val)==-1)
42958             return(-1);
42959         return(val);
42960     }
```

42961 The use of [ENOTTY] to indicate an incorrect descriptor type matches the historical behavior of
42962 SIOCATMARK.

42963 **FUTURE DIRECTIONS**

42964 None.

42965 **SEE ALSO**

42966 *pselect()*, *recv()*, *recvmsg()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**sys/socket.h**>

42967 **CHANGE HISTORY**

42968 First released in Issue 6. Derived from IEEE Std 1003.1g-2000.

42969 **NAME**

42970 socket — create an endpoint for communication

42971 **SYNOPSIS**

42972 #include <sys/socket.h>

42973 int socket(int *domain*, int *type*, int *protocol*);42974 **DESCRIPTION**42975 The *socket()* function shall create an unbound socket in a communications domain, and return a
42976 file descriptor that can be used in later function calls that operate on sockets.42977 The *socket()* function takes the following arguments:42978 *domain* Specifies the communications domain in which a socket is to be created.42979 *type* Specifies the type of socket to be created.42980 *protocol* Specifies a particular protocol to be used with the socket. Specifying a *protocol*
42981 of 0 causes *socket()* to use an unspecified default protocol appropriate for the
42982 requested socket type.42983 The *domain* argument specifies the address family used in the communications domain. The
42984 address families supported by the system are implementation-defined.42985 Symbolic constants that can be used for the domain argument are defined in the <sys/socket.h>
42986 header.42987 The *type* argument specifies the socket type, which determines the semantics of communication
42988 over the socket. The following socket types are defined; implementations may specify additional
42989 socket types:42990 SOCK_STREAM Provides sequenced, reliable, bidirectional, connection-mode byte
42991 streams, and may provide a transmission mechanism for out-of-band
42992 data.42993 SOCK_DGRAM Provides datagrams, which are connectionless-mode, unreliable messages
42994 of fixed maximum length.42995 SOCK_SEQPACKET Provides sequenced, reliable, bidirectional, connection-mode
42996 transmission paths for records. A record can be sent using one or more
42997 output operations and received using one or more input operations, but a
42998 single operation never transfers part of more than one record. Record
42999 boundaries are visible to the receiver via the MSG_EOR flag.43000 If the *protocol* argument is non-zero, it shall specify a protocol that is supported by the address
43001 family. If the *protocol* argument is zero, the default protocol for this address family and type shall
43002 be used. The protocols supported by the system are implementation-defined.43003 The process may need to have appropriate privileges to use the *socket()* function or to create
43004 some sockets.43005 **RETURN VALUE**43006 Upon successful completion, *socket()* shall return a non-negative integer, the socket file
43007 descriptor. Otherwise, a value of -1 shall be returned and *errno* set to indicate the error.43008 **ERRORS**43009 The *socket()* function shall fail if:

43010 [EAFNOSUPPORT]

43011 The implementation does not support the specified address family.

- 43012 [EMFILE] No more file descriptors are available for this process.
- 43013 [ENFILE] No more file descriptors are available for the system.
- 43014 [EPROTONOSUPPORT]
 43015 The protocol is not supported by the address family, or the protocol is not
 43016 supported by the implementation.
- 43017 [EPROTOTYPE] The socket type is not supported by the protocol.
- 43018 The *socket()* function may fail if:
- 43019 [EACCES] The process does not have appropriate privileges.
- 43020 [ENOBUFS] Insufficient resources were available in the system to perform the operation.
- 43021 [ENOMEM] Insufficient memory was available to fulfill the request.

43022 **EXAMPLES**

43023 None.

43024 **APPLICATION USAGE**

43025 The documentation for specific address families specifies which protocols each address family
 43026 supports. The documentation for specific protocols specifies which socket types each protocol
 43027 supports.

43028 The application can determine whether an address family is supported by trying to create a
 43029 socket with *domain* set to the protocol in question.

43030 **RATIONALE**

43031 None.

43032 **FUTURE DIRECTIONS**

43033 None.

43034 **SEE ALSO**

43035 *accept()*, *bind()*, *connect()*, *getsockname()*, *getsockopt()*, *listen()*, *recv()*, *recvfrom()*, *recvmsg()*,
 43036 *send()*, *sendmsg()*, *setsockopt()*, *shutdown()*, *socketpair()*, the Base Definitions volume of
 43037 IEEE Std 1003.1-2001, <netinet/in.h>, <sys/socket.h>

43038 **CHANGE HISTORY**

43039 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

43040 **NAME**

43041 socketpair — create a pair of connected sockets

43042 **SYNOPSIS**

43043 #include <sys/socket.h>

```
43044       int socketpair(int domain, int type, int protocol,
43045                      int socket_vector[2]);
```

43046 **DESCRIPTION**

43047 The *socketpair()* function shall create an unbound pair of connected sockets in a specified *domain*,
 43048 of a specified *type*, under the protocol optionally specified by the *protocol* argument. The two
 43049 sockets shall be identical. The file descriptors used in referencing the created sockets shall be
 43050 returned in *socket_vector*[0] and *socket_vector*[1].

43051 The *socketpair()* function takes the following arguments:

43052	<i>domain</i>	Specifies the communications domain in which the sockets are to be created.
43053	<i>type</i>	Specifies the type of sockets to be created.
43054	<i>protocol</i>	Specifies a particular protocol to be used with the sockets. Specifying a
43055		<i>protocol</i> of 0 causes <i>socketpair()</i> to use an unspecified default protocol
43056		appropriate for the requested socket type.
43057	<i>socket_vector</i>	Specifies a 2-integer array to hold the file descriptors of the created socket
43058		pair.

43059 The *type* argument specifies the socket type, which determines the semantics of communications
 43060 over the socket. The following socket types are defined; implementations may specify additional
 43061 socket types:

43062	SOCK_STREAM	Provides sequenced, reliable, bidirectional, connection-mode byte
43063		streams, and may provide a transmission mechanism for out-of-band
43064		data.
43065	SOCK_DGRAM	Provides datagrams, which are connectionless-mode, unreliable messages
43066		of fixed maximum length.
43067	SOCK_SEQPACKET	Provides sequenced, reliable, bidirectional, connection-mode
43068		transmission paths for records. A record can be sent using one or more
43069		output operations and received using one or more input operations, but a
43070		single operation never transfers part of more than one record. Record
43071		boundaries are visible to the receiver via the MSG_EOR flag.

43072 If the *protocol* argument is non-zero, it shall specify a protocol that is supported by the address
 43073 family. If the *protocol* argument is zero, the default protocol for this address family and type shall
 43074 be used. The protocols supported by the system are implementation-defined.

43075 The process may need to have appropriate privileges to use the *socketpair()* function or to create
 43076 some sockets.

43077 **RETURN VALUE**

43078 Upon successful completion, this function shall return 0; otherwise, -1 shall be returned and
 43079 *errno* set to indicate the error.

43080 **ERRORS**

43081 The *socketpair()* function shall fail if:

43082 [EAFNOSUPPORT]

43083 The implementation does not support the specified address family.

- 43084 [EMFILE] No more file descriptors are available for this process.
- 43085 [ENFILE] No more file descriptors are available for the system.
- 43086 [EOPNOTSUPP] The specified protocol does not permit creation of socket pairs.
- 43087 [EPROTONOSUPPORT]
- 43088 The protocol is not supported by the address family, or the protocol is not
- 43089 supported by the implementation.
- 43090 [EPROTOTYPE] The socket type is not supported by the protocol.
- 43091 The *socketpair()* function may fail if:
- 43092 [EACCES] The process does not have appropriate privileges.
- 43093 [ENOBUFS] Insufficient resources were available in the system to perform the operation.
- 43094 [ENOMEM] Insufficient memory was available to fulfill the request.

43095 EXAMPLES

43096 None.

43097 APPLICATION USAGE

43098 The documentation for specific address families specifies which protocols each address family
 43099 supports. The documentation for specific protocols specifies which socket types each protocol
 43100 supports.

43101 The *socketpair()* function is used primarily with UNIX domain sockets and need not be
 43102 supported for other domains.

43103 RATIONALE

43104 None.

43105 FUTURE DIRECTIONS

43106 None.

43107 SEE ALSO

43108 *socket()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>

43109 CHANGE HISTORY

43110 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

43111 **NAME**43112 `sprintf` — print formatted output43113 **SYNOPSIS**43114 `#include <stdio.h>`43115 `int sprintf(char *restrict s, const char *restrict format, ...);`43116 **DESCRIPTION**43117 Refer to `fprintf()`.

43118 **NAME**

43119 sqrt, sqrtf, sqrtl — square root function

43120 **SYNOPSIS**

43121 #include <math.h>

43122 double sqrt(double x);

43123 float sqrtf(float x);

43124 long double sqrtl(long double x);

43125 **DESCRIPTION**

43126 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 43127 conflict between the requirements described here and the ISO C standard is unintentional. This
 43128 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

43129 These functions shall compute the square root of their argument x , \sqrt{x} .

43130 An application wishing to check for error situations should set *errno* to zero and call
 43131 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 43132 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 43133 zero, an error has occurred.

43134 **RETURN VALUE**43135 Upon successful completion, these functions shall return the square root of x .

43136 MX For finite values of $x < -0$, a domain error shall occur, and either a NaN (if supported), or an
 43137 implementation-defined value shall be returned.

43138 MX If x is NaN, a NaN shall be returned.

43139 If x is ± 0 or $+\text{Inf}$, x shall be returned.

43140 If x is $-\text{Inf}$, a domain error shall occur, and either a NaN (if supported), or an implementation-
 43141 defined value shall be returned.

43142 **ERRORS**

43143 These functions shall fail if:

43144 MX Domain Error The finite value of x is < -0 , or x is $-\text{Inf}$.

43145 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 43146 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling
 43147 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 43148 shall be raised.

43149 **EXAMPLES**43150 **Taking the Square Root of 9.0**

43151 #include <math.h>

43152 ...

43153 double x = 9.0;

43154 double result;

43155 ...

43156 result = sqrt(x);

43157 APPLICATION USAGE

43158 On error, the expressions (`math_errhandling & MATH_ERRNO`) and (`math_errhandling &`
43159 `MATH_ERREXCEPT`) are independent of each other, but at least one of them must be non-zero.

43160 RATIONALE

43161 None.

43162 FUTURE DIRECTIONS

43163 None.

43164 SEE ALSO

43165 *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-2001,
43166 Section 4.18, Treatment of Error Conditions for Mathematical Functions, `<math.h>`, `<stdio.h>`

43167 CHANGE HISTORY

43168 First released in Issue 1. Derived from Issue 1 of the SVID.

43169 Issue 5

43170 The DESCRIPTION is updated to indicate how an application should check for an error. This
43171 text was previously published in the APPLICATION USAGE section.

43172 Issue 6

43173 The *sqrtrf()* and *sqrtrl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

43174 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
43175 revised to align with the ISO/IEC 9899:1999 standard.

43176 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
43177 marked.

43178 NAME

43179 srand — pseudo-random number generator

43180 SYNOPSIS

43181 #include <stdlib.h>

43182 void srand(unsigned *seed*);

43183 DESCRIPTION

43184 Refer to *rand()*.

43185 **NAME**

43186 srand48 — seed the uniformly distributed double-precision pseudo-random number generator

43187 **SYNOPSIS**

43188 xSI #include <stdlib.h>

43189 void srand48(long *seedval*);

43190

43191 **DESCRIPTION**

43192 Refer to *drand48()*.

43193 NAME

43194 srandom — seed pseudo-random number generator

43195 SYNOPSIS

43196 XSI #include <stdlib.h>

43197 void srandom(unsigned *seed*);

43198

43199 DESCRIPTION

43200 Refer to *initstate()*.

43201 **NAME**

43202 sscanf — convert formatted input

43203 **SYNOPSIS**

43204 #include <stdio.h>

43205 int sscanf(const char *restrict *s*, const char *restrict *format*, ...);43206 **DESCRIPTION**43207 Refer to *fscanf()*.

43208 NAME

43209 stat — get file status

43210 SYNOPSIS

43211 #include <sys/stat.h>

43212 int stat(const char *restrict path, struct stat *restrict buf);

43213 DESCRIPTION

43214 The *stat()* function shall obtain information about the named file and write it to the area pointed
 43215 to by the *buf* argument. The *path* argument points to a pathname naming a file. Read, write, or
 43216 execute permission of the named file is not required. An implementation that provides
 43217 additional or alternate file access control mechanisms may, under implementation-defined
 43218 conditions, cause *stat()* to fail. In particular, the system may deny the existence of the file
 43219 specified by *path*.

43220 If the named file is a symbolic link, the *stat()* function shall continue pathname resolution using
 43221 the contents of the symbolic link, and shall return information pertaining to the resulting file if
 43222 the file exists.

43223 The *buf* argument is a pointer to a **stat** structure, as defined in the <sys/stat.h> header, into
 43224 which information is placed concerning the file.

43225 The *stat()* function shall update any time-related fields (as described in the Base Definitions
 43226 volume of IEEE Std 1003.1-2001, Section 4.7, File Times Update), before writing into the **stat**
 43227 structure.

43228 Unless otherwise specified, the structure members *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atime*,
 43229 *st_ctime*, and *st_mtime* shall have meaningful values for all file types defined in this volume of
 43230 IEEE Std 1003.1-2001. The value of the member *st_nlink* shall be set to the number of links to the
 43231 file.

43232 RETURN VALUE

43233 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
 43234 indicate the error.

43235 ERRORS

43236 The *stat()* function shall fail if:

43237 [EACCES] Search permission is denied for a component of the path prefix.

43238 [EIO] An error occurred while reading from the file system.

43239 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 43240 argument.

43241 [ENAMETOOLONG]

43242 The length of the *path* argument exceeds {PATH_MAX} or a pathname
 43243 component is longer than {NAME_MAX}.

43244 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

43245 [ENOTDIR] A component of the path prefix is not a directory.

43246 [EOVERFLOW] The file size in bytes or the number of blocks allocated to the file or the file
 43247 serial number cannot be represented correctly in the structure pointed to by
 43248 *buf*.

43249 The *stat()* function may fail if:

43250 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 43251 resolution of the *path* argument.

43252 [ENAMETOOLONG]
 43253 As a result of encountering a symbolic link in resolution of the *path* argument,
 43254 the length of the substituted pathname string exceeded {PATH_MAX}.

43255 [EOVERFLOW] A value to be stored would overflow one of the members of the **stat** structure.

43256 EXAMPLES

43257 Obtaining File Status Information

43258 The following example shows how to obtain file status information for a file named
 43259 **/home/cnd/mod1**. The structure variable *buffer* is defined for the **stat** structure.

```
43260 #include <sys/types.h>
43261 #include <sys/stat.h>
43262 #include <fcntl.h>

43263 struct stat buffer;
43264 int      status;
43265 ...
43266 status = stat("/home/cnd/mod1", &buffer);
```

43267 Getting Directory Information

43268 The following example fragment gets status information for each entry in a directory. The call to
 43269 the *stat()* function stores file information in the **stat** structure pointed to by *statbuf*. The lines
 43270 that follow the *stat()* call format the fields in the **stat** structure for presentation to the user of the
 43271 program.

```
43272 #include <sys/types.h>
43273 #include <sys/stat.h>
43274 #include <dirent.h>
43275 #include <pwd.h>
43276 #include <grp.h>
43277 #include <time.h>
43278 #include <locale.h>
43279 #include <langinfo.h>
43280 #include <stdio.h>
43281 #include <stdint.h>

43282 struct dirent  *dp;
43283 struct stat    statbuf;
43284 struct passwd  *pwd;
43285 struct group   *grp;
43286 struct tm      *tm;
43287 char           datestring[256];
43288 ...
43289 /* Loop through directory entries. */
43290 while ((dp = readdir(dir)) != NULL) {
43291     /* Get entry's information. */
43292     if (stat(dp->d_name, &statbuf) == -1)
43293         continue;
```



```

43294      /* Print out type, permissions, and number of links. */
43295      printf("%10.10s", sperm (statbuf.st_mode));
43296      printf("%4d", statbuf.st_nlink);

43297      /* Print out owner's name if it is found using getpwuid(). */
43298      if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
43299          printf(" %-8.8s", pwd->pw_name);
43300      else
43301          printf(" %-8d", statbuf.st_uid);

43302      /* Print out group name if it is found using getgrgid(). */
43303      if ((grp = getgrgid(statbuf.st_gid)) != NULL)
43304          printf(" %-8.8s", grp->gr_name);
43305      else
43306          printf(" %-8d", statbuf.st_gid);

43307      /* Print size of file. */
43308      printf(" %9jd", (intmax_t)statbuf.st_size);

43309      tm = localtime(&statbuf.st_mtime);

43310      /* Get localized date string. */
43311      strftime(datestring, sizeof(datestring), nl_langinfo(D_T_FMT), tm);

43312      printf(" %s %s\n", datestring, dp->d_name);
43313  }

```

43314 APPLICATION USAGE

43315 None.

43316 RATIONALE

43317 The intent of the paragraph describing “additional or alternate file access control mechanisms”
 43318 is to allow a secure implementation where a process with a label that does not dominate the
 43319 file’s label cannot perform a *stat()* function. This is not related to read permission; a process with
 43320 a label that dominates the file’s label does not need read permission. An implementation that
 43321 supports write-up operations could fail *fstat()* function calls even though it has a valid file
 43322 descriptor open for writing.

43323 FUTURE DIRECTIONS

43324 None.

43325 SEE ALSO

43326 *fstat()*, *lstat()*, *readlink()*, *symlink()*, the Base Definitions volume of IEEE Std 1003.1-2001,
 43327 **<sys/stat.h>**, **<sys/types.h>**

43328 CHANGE HISTORY

43329 First released in Issue 1. Derived from Issue 1 of the SVID.

43330 Issue 5

43331 Large File Summit extensions are added.

43332 Issue 6

43333 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

43334 The following new requirements on POSIX implementations derive from alignment with the
 43335 Single UNIX Specification:

- 43336 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
 43337 required for conforming implementations of previous POSIX specifications, it was not
 43338 required for UNIX applications.

- 43339 • The [EIO] mandatory error condition is added.
- 43340 • The [ELOOP] mandatory error condition is added.
- 43341 • The [EOVERFLOW] mandatory error condition is added. This change is to support large
- 43342 files.
- 43343 • The [ENAMETOOLONG] and the second [EOVERFLOW] optional error conditions are
- 43344 added.
- 43345 The following changes were made to align with the IEEE P1003.1a draft standard:
- 43346 • Details are added regarding the treatment of symbolic links.
- 43347 • The [ELOOP] optional error condition is added.
- 43348 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
- 43349 The **restrict** keyword is added to the *stat()* prototype for alignment with the ISO/IEC 9899:1999
- 43350 standard.

43351 NAME

43352 statvfs — get file system information

43353 SYNOPSIS

43354 XSI #include <sys/statvfs.h>

43355 int statvfs(const char *restrict *path*, struct statvfs *restrict *buf*);

43356

43357 DESCRIPTION

43358 Refer to *fstatvfs()*.

43359 **NAME**

43360 stderr, stdin, stdout — standard I/O streams

43361 **SYNOPSIS**

43362 #include <stdio.h>

43363 extern FILE *stderr, *stdin, *stdout;

43364 **DESCRIPTION**

43365 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 43366 conflict between the requirements described here and the ISO C standard is unintentional. This
 43367 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

43368 A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type
 43369 **FILE**. The *fopen()* function shall create certain descriptive data for a stream and return a pointer
 43370 to designate the stream in all further transactions. Normally, there are three open streams with
 43371 constant pointers declared in the <stdio.h> header and associated with the standard open files.

43372 At program start-up, three streams shall be predefined and need not be opened explicitly:
 43373 *standard input* (for reading conventional input), *standard output* (for writing conventional output),
 43374 and *standard error* (for writing diagnostic output). When opened, the standard error stream is not
 43375 fully buffered; the standard input and standard output streams are fully buffered if and only if
 43376 the stream can be determined not to refer to an interactive device.

43377 CX The following symbolic values in <unistd.h> define the file descriptors that shall be associated
 43378 with the C-language *stdin*, *stdout*, and *stderr* when the application is started:

43379 STDIN_FILENO Standard input value, *stdin*. Its value is 0.

43380 STDOUT_FILENO Standard output value, *stdout*. Its value is 1.

43381 STDERR_FILENO Standard error value, *stderr*. Its value is 2.

43382 The *stderr* stream is expected to be open for reading and writing.

43383 **RETURN VALUE**

43384 None.

43385 **ERRORS**

43386 No errors are defined.

43387 **EXAMPLES**

43388 None.

43389 **APPLICATION USAGE**

43390 None.

43391 **RATIONALE**

43392 None.

43393 **FUTURE DIRECTIONS**

43394 None.

43395 **SEE ALSO**

43396 *fclose()*, *feof()*, *ferror()*, *fileno()*, *fopen()*, *fread()*, *fseek()*, *getc()*, *gets()*, *popen()*, *printf()*, *putc()*,
 43397 *puts()*, *read()*, *scanf()*, *setbuf()*, *setvbuf()*, *tmpfile()*, *ungetc()*, *vprintf()*, the Base Definitions
 43398 volume of IEEE Std 1003.1-2001, <stdio.h>, <unistd.h>

43399 **CHANGE HISTORY**

43400 First released in Issue 1.

43401 **Issue 6**

43402 Extensions beyond the ISO C standard are marked.

43403 A note that *stderr* is expected to be open for reading and writing is added to the DESCRIPTION.

43404 **NAME**

43405 strcasecmp, strncasecmp — case-insensitive string comparisons

43406 **SYNOPSIS**

43407 XSI #include <strings.h>

43408 int strcasecmp(const char *s1, const char *s2);

43409 int strncasecmp(const char *s1, const char *s2, size_t n);

43410

43411 **DESCRIPTION**

43412 The *strcasecmp()* function shall compare, while ignoring differences in case, the string pointed to
 43413 by *s1* to the string pointed to by *s2*. The *strncasecmp()* function shall compare, while ignoring
 43414 differences in case, not more than *n* bytes from the string pointed to by *s1* to the string pointed to
 43415 by *s2*.

43416 In the POSIX locale, *strcasecmp()* and *strncasecmp()* shall behave as if the strings had been
 43417 converted to lowercase and then a byte comparison performed. The results are unspecified in
 43418 other locales.

43419 **RETURN VALUE**

43420 Upon completion, *strcasecmp()* shall return an integer greater than, equal to, or less than 0, if the
 43421 string pointed to by *s1* is, ignoring case, greater than, equal to, or less than the string pointed to
 43422 by *s2*, respectively.

43423 Upon successful completion, *strncasecmp()* shall return an integer greater than, equal to, or less
 43424 than 0, if the possibly null-terminated array pointed to by *s1* is, ignoring case, greater than, equal
 43425 to, or less than the possibly null-terminated array pointed to by *s2*, respectively.

43426 **ERRORS**

43427 No errors are defined.

43428 **EXAMPLES**

43429 None.

43430 **APPLICATION USAGE**

43431 None.

43432 **RATIONALE**

43433 None.

43434 **FUTURE DIRECTIONS**

43435 None.

43436 **SEE ALSO**

43437 The Base Definitions volume of IEEE Std 1003.1-2001, <strings.h>

43438 **CHANGE HISTORY**

43439 First released in Issue 4, Version 2.

43440 **Issue 5**

43441 Moved from X/OPEN UNIX extension to BASE.

43442 **NAME**

43443 strcat — concatenate two strings

43444 **SYNOPSIS**

43445 #include <string.h>

43446 char *strcat(char *restrict *s1*, const char *restrict *s2*);43447 **DESCRIPTION**

43448 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
43449 conflict between the requirements described here and the ISO C standard is unintentional. This
43450 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

43451 The *strcat()* function shall append a copy of the string pointed to by *s2* (including the
43452 terminating null byte) to the end of the string pointed to by *s1*. The initial byte of *s2* overwrites
43453 the null byte at the end of *s1*. If copying takes place between objects that overlap, the behavior is
43454 undefined.

43455 **RETURN VALUE**43456 The *strcat()* function shall return *s1*; no return value is reserved to indicate an error.43457 **ERRORS**

43458 No errors are defined.

43459 **EXAMPLES**

43460 None.

43461 **APPLICATION USAGE**

43462 This issue is aligned with the ISO C standard; this does not affect compatibility with XPG3
43463 applications. Reliable error detection by this function was never guaranteed.

43464 **RATIONALE**

43465 None.

43466 **FUTURE DIRECTIONS**

43467 None.

43468 **SEE ALSO**43469 *strncat()*, the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>43470 **CHANGE HISTORY**

43471 First released in Issue 1. Derived from Issue 1 of the SVID.

43472 **Issue 6**43473 The *strcat()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

43474 **NAME**

43475 strchr — string scanning operation

43476 **SYNOPSIS**

43477 #include <string.h>

43478 char *strchr(const char *s, int c);

43479 **DESCRIPTION**

43480 cx The functionality described on this reference page is aligned with the ISO C standard. Any
43481 conflict between the requirements described here and the ISO C standard is unintentional. This
43482 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

43483 The *strchr()* function shall locate the first occurrence of *c* (converted to a **char**) in the string
43484 pointed to by *s*. The terminating null byte is considered to be part of the string.

43485 **RETURN VALUE**

43486 Upon completion, *strchr()* shall return a pointer to the byte, or a null pointer if the byte was not
43487 found.

43488 **ERRORS**

43489 No errors are defined.

43490 **EXAMPLES**

43491 None.

43492 **APPLICATION USAGE**

43493 None.

43494 **RATIONALE**

43495 None.

43496 **FUTURE DIRECTIONS**

43497 None.

43498 **SEE ALSO**43499 *strrchr()*, the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>43500 **CHANGE HISTORY**

43501 First released in Issue 1. Derived from Issue 1 of the SVID.

43502 **Issue 6**

43503 Extensions beyond the ISO C standard are marked.

43504 NAME

43505 strcmp — compare two strings

43506 SYNOPSIS

43507 #include <string.h>

43508 int strcmp(const char *s1, const char *s2);

43509 DESCRIPTION

43510 cx The functionality described on this reference page is aligned with the ISO C standard. Any
 43511 conflict between the requirements described here and the ISO C standard is unintentional. This
 43512 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

43513 The *strcmp()* function shall compare the string pointed to by *s1* to the string pointed to by *s2*.

43514 The sign of a non-zero return value shall be determined by the sign of the difference between the
 43515 values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the strings
 43516 being compared.

43517 RETURN VALUE

43518 Upon completion, *strcmp()* shall return an integer greater than, equal to, or less than 0, if the
 43519 string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2*,
 43520 respectively.

43521 ERRORS

43522 No errors are defined.

43523 EXAMPLES

43524 Checking a Password Entry

43525 The following example compares the information read from standard input to the value of the
 43526 name of the user entry. If the *strcmp()* function returns 0 (indicating a match), a further check
 43527 will be made to see if the user entered the proper old password. The *crypt()* function shall
 43528 encrypt the old password entered by the user, using the value of the encrypted password in the
 43529 **passwd** structure as the salt. If this value matches the value of the encrypted **passwd** in the
 43530 structure, the entered password *oldpasswd* is the correct user's password. Finally, the program
 43531 encrypts the new password so that it can store the information in the **passwd** structure.

```

43532 #include <string.h>
43533 #include <unistd.h>
43534 #include <stdio.h>
43535 ...
43536 int valid_change;
43537 struct passwd *p;
43538 char user[100];
43539 char oldpasswd[100];
43540 char newpasswd[100];
43541 char savepasswd[100];
43542 ...
43543 if (strcmp(p->pw_name, user) == 0) {
43544     if (strcmp(p->pw_passwd, crypt(oldpasswd, p->pw_passwd)) == 0) {
43545         strcpy(savepasswd, crypt(newpasswd, user));
43546         p->pw_passwd = savepasswd;
43547         valid_change = 1;
43548     }
43549     else {
```



```
43550         fprintf(stderr, "Old password is not valid\n");
43551     }
43552 }
43553 ...
```

43554 APPLICATION USAGE

43555 None.

43556 RATIONALE

43557 None.

43558 FUTURE DIRECTIONS

43559 None.

43560 SEE ALSO

43561 *strncmp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**string.h**>

43562 CHANGE HISTORY

43563 First released in Issue 1. Derived from Issue 1 of the SVID.

43564 Issue 6

43565 Extensions beyond the ISO C standard are marked.

43566 **NAME**

43567 strcoll — string comparison using collating information

43568 **SYNOPSIS**

43569 #include <string.h>

43570 int strcoll(const char *s1, const char *s2);

43571 **DESCRIPTION**

43572 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 43573 conflict between the requirements described here and the ISO C standard is unintentional. This
 43574 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

43575 The *strcoll()* function shall compare the string pointed to by *s1* to the string pointed to by *s2*,
 43576 both interpreted as appropriate to the *LC_COLLATE* category of the current locale.

43577 CX The *strcoll()* function shall not change the setting of *errno* if successful.

43578 Since no return value is reserved to indicate an error, an application wishing to check for error
 43579 situations should set *errno* to 0, then call *strcoll()*, then check *errno*.

43580 **RETURN VALUE**

43581 Upon successful completion, *strcoll()* shall return an integer greater than, equal to, or less than 0,
 43582 according to whether the string pointed to by *s1* is greater than, equal to, or less than the string
 43583 CX pointed to by *s2* when both are interpreted as appropriate to the current locale. On error,
 43584 *strcoll()* may set *errno*, but no return value is reserved to indicate an error.

43585 **ERRORS**43586 The *strcoll()* function may fail if:

43587 CX [EINVAL] The *s1* or *s2* arguments contain characters outside the domain of the collating
 43588 sequence.

43589 **EXAMPLES**43590 **Comparing Nodes**

43591 The following example uses an application-defined function, *node_compare()*, to compare two
 43592 nodes based on an alphabetical ordering of the *string* field.

```
43593 #include <string.h>
43594 ...
43595 struct node { /* These are stored in the table. */
43596     char *string;
43597     int length;
43598 };
43599 ...
43600 int node_compare(const void *node1, const void *node2)
43601 {
43602     return strcoll(((const struct node *)node1)->string,
43603                   ((const struct node *)node2)->string);
43604 }
43605 ...
```

43606 **APPLICATION USAGE**43607 The *strxfrm()* and *strcmp()* functions should be used for sorting large lists.

43608 **RATIONALE**

43609 None.

43610 **FUTURE DIRECTIONS**

43611 None.

43612 **SEE ALSO**43613 *strcmp()*, *strxfrm()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**string.h**>43614 **CHANGE HISTORY**

43615 First released in Issue 3.

43616 **Issue 5**43617 The DESCRIPTION is updated to indicate that *errno* does not change if the function is
43618 successful.43619 **Issue 6**

43620 Extensions beyond the ISO C standard are marked.

43621 The following new requirements on POSIX implementations derive from alignment with the
43622 Single UNIX Specification:

- 43623
- The [EINVAL] optional error condition is added.

43624 An example is added.

43625 **NAME**

43626 strcpy — copy a string

43627 **SYNOPSIS**

43628 #include <string.h>

43629 char *strcpy(char *restrict s1, const char *restrict s2);

43630 **DESCRIPTION**

43631 cx The functionality described on this reference page is aligned with the ISO C standard. Any
 43632 conflict between the requirements described here and the ISO C standard is unintentional. This
 43633 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

43634 The *strcpy()* function shall copy the string pointed to by *s2* (including the terminating null byte)
 43635 into the array pointed to by *s1*. If copying takes place between objects that overlap, the behavior
 43636 is undefined.

43637 **RETURN VALUE**43638 The *strcpy()* function shall return *s1*; no return value is reserved to indicate an error.43639 **ERRORS**

43640 No errors are defined.

43641 **EXAMPLES**43642 **Initializing a String**43643 The following example copies the string "-----" into the *permstring* variable.

```
43644       #include <string.h>
43645       ...
43646       static char permstring[11];
43647       ...
43648       strcpy(permstring, "-----");
43649       ...
```

43650 **Storing a Key and Data**

43651 The following example allocates space for a key using *malloc()* then uses *strcpy()* to place the
 43652 key there. Then it allocates space for data using *malloc()*, and uses *strcpy()* to place data there.
 43653 (The user-defined function *dbfree()* frees memory previously allocated to an array of type **struct**
 43654 **element** *.)

```
43655       #include <string.h>
43656       #include <stdlib.h>
43657       #include <stdio.h>
43658       ...
43659       /* Structure used to read data and store it. */
43660       struct element {
43661           char *key;
43662           char *data;
43663       };
43664       struct element *tbl, *curtbl;
43665       char *key, *data;
43666       int count;
43667       ...
43668       void dbfree(struct element *, int);
```



```
43669     ...
43670     if ((curtbl->key = malloc(strlen(key) + 1)) == NULL) {
43671         perror("malloc"); dbfree(tbl, count); return NULL;
43672     }
43673     strcpy(curtbl->key, key);
43674     if ((curtbl->data = malloc(strlen(data) + 1)) == NULL) {
43675         perror("malloc"); free(curtbl->key); dbfree(tbl, count); return NULL;
43676     }
43677     strcpy(curtbl->data, data);
43678     ...
```

43679 APPLICATION USAGE

43680 Character movement is performed differently in different implementations. Thus, overlapping
43681 moves may yield surprises.

43682 This issue is aligned with the ISO C standard; this does not affect compatibility with XPG3
43683 applications. Reliable error detection by this function was never guaranteed.

43684 RATIONALE

43685 None.

43686 FUTURE DIRECTIONS

43687 None.

43688 SEE ALSO

43689 *strncpy()*, the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

43690 CHANGE HISTORY

43691 First released in Issue 1. Derived from Issue 1 of the SVID.

43692 Issue 6

43693 The *strcpy()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

43694 **NAME**

43695 strcspn — get the length of a complementary substring

43696 **SYNOPSIS**

43697 #include <string.h>

43698 size_t strcspn(const char *s1, const char *s2);

43699 **DESCRIPTION**

43700 cx The functionality described on this reference page is aligned with the ISO C standard. Any
43701 conflict between the requirements described here and the ISO C standard is unintentional. This
43702 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

43703 The *strcspn()* function shall compute the length (in bytes) of the maximum initial segment of the
43704 string pointed to by *s1* which consists entirely of bytes *not* from the string pointed to by *s2*.

43705 **RETURN VALUE**

43706 The *strcspn()* function shall return the length of the computed segment of the string pointed to
43707 by *s1*; no return value is reserved to indicate an error.

43708 **ERRORS**

43709 No errors are defined.

43710 **EXAMPLES**

43711 None.

43712 **APPLICATION USAGE**

43713 None.

43714 **RATIONALE**

43715 None.

43716 **FUTURE DIRECTIONS**

43717 None.

43718 **SEE ALSO**43719 *strspn()*, the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>43720 **CHANGE HISTORY**

43721 First released in Issue 1. Derived from Issue 1 of the SVID.

43722 **Issue 5**

43723 The RETURN VALUE section is updated to indicate that *strcspn()* returns the length of *s1*, and
43724 not *s1* itself as was previously stated.

43725 **Issue 6**

43726 The Open Group Corrigendum U030/1 is applied. The text of the RETURN VALUE section is
43727 updated to indicate that the computed segment length is returned, not the *s1* length.

43728 **NAME**

43729 strdup — duplicate a string

43730 **SYNOPSIS**

43731 XSI #include <string.h>

43732 char *strdup(const char *s1);

43733

43734 **DESCRIPTION**

43735 The *strdup()* function shall return a pointer to a new string, which is a duplicate of the string
43736 pointed to by *s1*. The returned pointer can be passed to *free()*. A null pointer is returned if the
43737 new string cannot be created.

43738 **RETURN VALUE**

43739 The *strdup()* function shall return a pointer to a new string on success. Otherwise, it shall return
43740 a null pointer and set *errno* to indicate the error.

43741 **ERRORS**43742 The *strdup()* function may fail if:

43743 [ENOMEM] Storage space available is insufficient.

43744 **EXAMPLES**

43745 None.

43746 **APPLICATION USAGE**

43747 None.

43748 **RATIONALE**

43749 None.

43750 **FUTURE DIRECTIONS**

43751 None.

43752 **SEE ALSO**43753 *free()*, *malloc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>43754 **CHANGE HISTORY**

43755 First released in Issue 4, Version 2.

43756 **Issue 5**

43757 Moved from X/OPEN UNIX extension to BASE.

43758 NAME

43759 strerror, strerror_r — get error message string

43760 SYNOPSIS

43761 #include <string.h>

43762 char *strerror(int *errnum*);43763 TSF int strerror_r(int *errnum*, char **strerrbuf*, size_t *buflen*);

43764

43765 DESCRIPTION

43766 CX For *strerror()*: The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

43769 The *strerror()* function shall map the error number in *errnum* to a locale-dependent error message string and shall return a pointer to it. Typically, the values for *errnum* come from *errno*, but *strerror()* shall map any value of type **int** to a message.

43772 The string pointed to shall not be modified by the application, but may be overwritten by a subsequent call to *strerror()* or *perror()*.

43774 CX The contents of the error message strings returned by *strerror()* should be determined by the setting of the *LC_MESSAGES* category in the current locale.

43776 The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-2001 calls *strerror()*.

43778 CX The *strerror()* function shall not change the setting of *errno* if successful.

43779 Since no return value is reserved to indicate an error, an application wishing to check for error situations should set *errno* to 0, then call *strerror()*, then check *errno*.

43781 The *strerror()* function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

43783 TSF The *strerror_r()* function shall map the error number in *errnum* to a locale-dependent error message string and shall return the string in the buffer pointed to by *strerrbuf*, with length *buflen*.

43785

43786 RETURN VALUE

43787 Upon successful completion, *strerror()* shall return a pointer to the generated message string. On error *errno* may be set, but no return value is reserved to indicate an error.

43789 TSF Upon successful completion, *strerror_r()* shall return 0. Otherwise, an error number shall be returned to indicate the error.

43791 ERRORS

43792 These functions may fail if:

43793 [EINVAL] The value of *errnum* is not a valid error number.

43794 The *strerror_r()* function may fail if:

43795 TSF [ERANGE] Insufficient storage was supplied via *strerrbuf* and *buflen* to contain the generated message string.

43796

43797 **EXAMPLES**

43798 None.

43799 **APPLICATION USAGE**

43800 None.

43801 **RATIONALE**

43802 None.

43803 **FUTURE DIRECTIONS**

43804 None.

43805 **SEE ALSO**43806 *perror()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**string.h**>43807 **CHANGE HISTORY**

43808 First released in Issue 3.

43809 **Issue 5**43810 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

43811 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

43812 **Issue 6**

43813 Extensions beyond the ISO C standard are marked.

43814 The following new requirements on POSIX implementations derive from alignment with the
43815 Single UNIX Specification:43816

- In the RETURN VALUE section, the fact that *errno* may be set is added.

43817

- The [EINVAL] optional error condition is added.

43818 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

43819 The *strerror_r()* function is added in response to IEEE PASC Interpretation 1003.1c #39.43820 The *strerror_r()* function is marked as part of the Thread-Safe Functions option.

43821 **NAME**

43822 strfmon — convert monetary value to a string

43823 **SYNOPSIS**43824 XSI

```
#include <monetary.h>
```

```
43825      ssize_t strfmon(char *restrict s, size_t maxsize,
43826                     const char *restrict format, ...);
```

43827

43828 **DESCRIPTION**

43829 The *strfmon()* function shall place characters into the array pointed to by *s* as controlled by the
 43830 string pointed to by *format*. No more than *maxsize* bytes are placed into the array.

43831 The format is a character string, beginning and ending in its initial state, if any, that contains two
 43832 types of objects: *plain characters*, which are simply copied to the output stream, and *conversion*
 43833 *specifications*, each of which shall result in the fetching of zero or more arguments which are
 43834 converted and formatted. The results are undefined if there are insufficient arguments for the
 43835 format. If the format is exhausted while arguments remain, the excess arguments are simply
 43836 ignored.

43837 The application shall ensure that a conversion specification consists of the following sequence:

- 43838 • A '%' character
- 43839 • Optional flags
- 43840 • Optional field width
- 43841 • Optional left precision
- 43842 • Optional right precision
- 43843 • A required conversion specifier character that determines the conversion to be performed

43844 **Flags**

43845 One or more of the following optional flags can be specified to control the conversion:

- 43846 =*f* An '=' followed by a single character *f* which is used as the numeric fill character. In
 43847 order to work with precision or width counts, the fill character shall be a single byte
 43848 character; if not, the behavior is undefined. The default numeric fill character is the
 43849 <space>. This flag does not affect field width filling which always uses the <space>.
 43850 This flag is ignored unless a left precision (see below) is specified.
- 43851 ^ Do not format the currency amount with grouping characters. The default is to insert
 43852 the grouping characters if defined for the current locale.
- 43853 + or (Specify the style of representing positive and negative currency amounts. Only one of
 43854 '+' or '(' may be specified. If '+' is specified, the locale's equivalent of '+' and '-'
 43855 are used (for example, in the U.S., the empty string if positive and '-' if negative). If
 43856 '(' is specified, negative amounts are enclosed within parentheses. If neither flag is
 43857 specified, the '+' style is used.
- 43858 ! Suppress the currency symbol from the output conversion.
- 43859 - Specify the alignment. If this flag is present the result of the conversion is left-justified
 43860 (padded to the right) rather than right-justified. This flag shall be ignored unless a field
 43861 width (see below) is specified.

43862 **Field Width**

43863 *w* A decimal digit string *w* specifying a minimum field width in bytes in which the result
 43864 of the conversion is right-justified (or left-justified if the flag ‘-’ is specified). The
 43865 default is 0.

43866 **Left Precision**

43867 *#n* A ‘#’ followed by a decimal digit string *n* specifying a maximum number of digits
 43868 expected to be formatted to the left of the radix character. This option can be used to
 43869 keep the formatted output from multiple calls to the *strfmon()* function aligned in the
 43870 same columns. It can also be used to fill unused positions with a special character as in
 43871 "\$***123.45". This option causes an amount to be formatted as if it has the number
 43872 of digits specified by *n*. If more than *n* digit positions are required, this conversion
 43873 specification is ignored. Digit positions in excess of those actually required are filled
 43874 with the numeric fill character (see the *=f* flag above).

43875 If grouping has not been suppressed with the ‘^’ flag, and it is defined for the current
 43876 locale, grouping separators are inserted before the fill characters (if any) are added.
 43877 Grouping separators are not applied to fill characters even if the fill character is a digit.

43878 To ensure alignment, any characters appearing before or after the number in the
 43879 formatted output such as currency or sign symbols are padded as necessary with
 43880 <space>s to make their positive and negative formats an equal length.

43881 **Right Precision**

43882 *.p* A period followed by a decimal digit string *p* specifying the number of digits after the
 43883 radix character. If the value of the right precision *p* is 0, no radix character appears. If a
 43884 right precision is not included, a default specified by the current locale is used. The
 43885 amount being formatted is rounded to the specified number of digits prior to
 43886 formatting.

43887 **Conversion Specifier Characters**

43888 The conversion specifier characters and their meanings are:

43889 *i* The **double** argument is formatted according to the locale's international currency
 43890 format (for example, in the U.S.: USD 1,234.56). If the argument is $\pm\text{Inf}$ or NaN, the
 43891 result of the conversion is unspecified.

43892 *n* The **double** argument is formatted according to the locale's national currency format
 43893 (for example, in the U.S.: \$1,234.56). If the argument is $\pm\text{Inf}$ or NaN, the result of the
 43894 conversion is unspecified.

43895 *%* Convert to a ‘%’; no argument is converted. The entire conversion specification shall
 43896 be %%.

43897 **Locale Information**

43898 The *LC_MONETARY* category of the program's locale affects the behavior of this function
 43899 including the monetary radix character (which may be different from the numeric radix
 43900 character affected by the *LC_NUMERIC* category), the grouping separator, the currency
 43901 symbols, and formats. The international currency symbol should be conformant with the
 43902 ISO 4217:1995 standard.

43903 If the value of *maxsize* is greater than {SSIZE_MAX}, the result is implementation-defined.

43904 **RETURN VALUE**

43905 If the total number of resulting bytes including the terminating null byte is not more than
 43906 *maxsize*, *strfmon()* shall return the number of bytes placed into the array pointed to by *s*, not
 43907 including the terminating null byte. Otherwise, *-1* shall be returned, the contents of the array are
 43908 unspecified, and *errno* shall be set to indicate the error.

43909 **ERRORS**

43910 The *strfmon()* function shall fail if:

43911 [E2BIG] Conversion stopped due to lack of space in the buffer.

43912 **EXAMPLES**

43913 Given a locale for the U.S. and the values 123.45, *-123.45*, and 3456.781, the following output
 43914 might be produced. Square brackets ("*[]*") are used in this example to delimit the output.

43915	%n	[\$123.45]	Default formatting
43916		[-\$123.45]	
43917		[\$3,456.78]	
43918	%11n	[\$123.45]	Right align within an 11-character field
43919		[-\$123.45]	
43920		[\$3,456.78]	
43921	%#5n	[\$ 123.45]	Aligned columns for values up to 99 999
43922		[-\$ 123.45]	
43923		[\$ 3,456.78]	
43924	%=*#5n	[\$***123.45]	Specify a fill character
43925		[-\$***123.45]	
43926		[\$*3,456.78]	
43927	%=0#5n	[\$000123.45]	Fill characters do not use grouping
43928		[-\$000123.45]	even if the fill character is a digit
43929		[\$03,456.78]	
43930	%^#5n	[\$ 123.45]	Disable the grouping separator
43931		[-\$ 123.45]	
43932		[\$ 3456.78]	
43933	%^#5.0n	[\$ 123]	Round off to whole units
43934		[-\$ 123]	
43935		[\$ 3457]	
43936	%^#5.4n	[\$ 123.4500]	Increase the precision
43937		[-\$ 123.4500]	
43938		[\$ 3456.7810]	
43939	%(#5n	[\$ 123.45]	Use an alternative pos/neg style
43940		[(\$ 123.45)]	
43941		[\$ 3,456.78]	
43942	%!(#5n	[123.45]	Disable the currency symbol
43943		[(123.45)]	
43944		[3,456.78]	
43945	%-14#5.4n	[\$ 123.4500]	Left-justify the output
43946		[-\$ 123.4500]	
43947		[\$ 3,456.7810]	

43948 %14#5.4n [\$ 123.4500] Corresponding right-justified output
 43949 [-\$ 123.4500]
 43950 [\$ 3,456.7810]

43951 See also the EXAMPLES section in *fprintf()*.

43952 APPLICATION USAGE

43953 None.

43954 RATIONALE

43955 None.

43956 FUTURE DIRECTIONS

43957 Lowercase conversion characters are reserved for future standards use and uppercase for
 43958 implementation-defined use.

43959 SEE ALSO

43960 *fprintf()*, *localeconv()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**monetary.h**>

43961 CHANGE HISTORY

43962 First released in Issue 4.

43963 Issue 5

43964 Moved from ENHANCED I18N to BASE.

43965 The [ENOSYS] error is removed.

43966 A sentence is added to the DESCRIPTION warning about values of *maxsize* that are greater than
 43967 {SSIZE_MAX}.

43968 Issue 6

43969 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

43970 The **restrict** keyword is added to the *strfmon()* prototype for alignment with the
 43971 ISO/IEC 9899:1999 standard.

43972 The EXAMPLES section is reworked, clarifying the output format.

43973 NAME

43974 strftime — convert date and time to a string

43975 SYNOPSIS

43976 #include <time.h>

43977 size_t strftime(char *restrict *s*, size_t *maxsize*,43978 const char *restrict *format*, const struct tm *restrict *timeptr*);

43979 DESCRIPTION

43980 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 43981 conflict between the requirements described here and the ISO C standard is unintentional. This
 43982 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

43983 The *strftime()* function shall place bytes into the array pointed to by *s* as controlled by the string
 43984 pointed to by *format*. The format is a character string, beginning and ending in its initial shift
 43985 state, if any. The *format* string consists of zero or more conversion specifications and ordinary
 43986 characters. A conversion specification consists of a '%' character, possibly followed by an E or O
 43987 modifier, and a terminating conversion specifier character that determines the conversion
 43988 specification's behavior. All ordinary characters (including the terminating null byte) are copied
 43989 unchanged into the array. If copying takes place between objects that overlap, the behavior is
 43990 undefined. No more than *maxsize* bytes are placed into the array. Each conversion specifier is
 43991 replaced by appropriate characters as described in the following list. The appropriate characters
 43992 are determined using the *LC_TIME* category of the current locale and by the values of zero or
 43993 more members of the broken-down time structure pointed to by *timeptr*, as specified in brackets
 43994 in the description. If any of the specified values are outside the normal range, the characters
 43995 stored are unspecified.

43996 CX Local timezone information is used as though *strftime()* called *tzset()*.

43997 The following conversion specifications are supported:

43998	%a	Replaced by the locale's abbreviated weekday name. [<i>tm_wday</i>]
43999	%A	Replaced by the locale's full weekday name. [<i>tm_wday</i>]
44000	%b	Replaced by the locale's abbreviated month name. [<i>tm_mon</i>]
44001	%B	Replaced by the locale's full month name. [<i>tm_mon</i>]
44002	%c	Replaced by the locale's appropriate date and time representation. (See the Base
44003		Definitions volume of IEEE Std 1003.1-2001, <time.h>.)
44004	%C	Replaced by the year divided by 100 and truncated to an integer, as a decimal number
44005		[00,99]. [<i>tm_year</i>]
44006	%d	Replaced by the day of the month as a decimal number [01,31]. [<i>tm_mday</i>]
44007	%D	Equivalent to %m/%d/%Y. [<i>tm_mon</i> , <i>tm_mday</i> , <i>tm_year</i>]
44008	%e	Replaced by the day of the month as a decimal number [1,31]; a single digit is preceded
44009		by a space. [<i>tm_mday</i>]
44010	%F	Equivalent to %Y-%m-%d (the ISO 8601:2000 standard date format). [<i>tm_year</i> , <i>tm_mon</i> ,
44011		<i>tm_mday</i>]
44012	%g	Replaced by the last 2 digits of the week-based year (see below) as a decimal number
44013		[00,99]. [<i>tm_year</i> , <i>tm_wday</i> , <i>tm_yday</i>]
44014	%G	Replaced by the week-based year (see below) as a decimal number (for example, 1977).
44015		[<i>tm_year</i> , <i>tm_wday</i> , <i>tm_yday</i>]

44016	%h	Equivalent to %b. [<i>tm_mon</i>]
44017	%H	Replaced by the hour (24-hour clock) as a decimal number [00,23]. [<i>tm_hour</i>]
44018	%I	Replaced by the hour (12-hour clock) as a decimal number [01,12]. [<i>tm_hour</i>]
44019	%j	Replaced by the day of the year as a decimal number [001,366]. [<i>tm_yday</i>]
44020	%m	Replaced by the month as a decimal number [01,12]. [<i>tm_mon</i>]
44021	%M	Replaced by the minute as a decimal number [00,59]. [<i>tm_min</i>]
44022	%n	Replaced by a <newline>.
44023	%p	Replaced by the locale's equivalent of either a.m. or p.m. [<i>tm_hour</i>]
44024 CX	%r	Replaced by the time in a.m. and p.m. notation; in the POSIX locale this shall be
44025		equivalent to %I:%M:%S %p. [<i>tm_hour</i> , <i>tm_min</i> , <i>tm_sec</i>]
44026	%R	Replaced by the time in 24-hour notation (%H:%M). [<i>tm_hour</i> , <i>tm_min</i>]
44027	%S	Replaced by the second as a decimal number [00,60]. [<i>tm_sec</i>]
44028	%t	Replaced by a <tab>.
44029	%T	Replaced by the time (%H:%M:%S). [<i>tm_hour</i> , <i>tm_min</i> , <i>tm_sec</i>]
44030	%u	Replaced by the weekday as a decimal number [1,7], with 1 representing Monday.
44031		[<i>tm_wday</i>]
44032	%U	Replaced by the week number of the year as a decimal number [00,53]. The first
44033		Sunday of January is the first day of week 1; days in the new year before this are in
44034		week 0. [<i>tm_year</i> , <i>tm_wday</i> , <i>tm_yday</i>]
44035	%V	Replaced by the week number of the year (Monday as the first day of the week) as a
44036		decimal number [01,53]. If the week containing 1 January has four or more days in the
44037		new year, then it is considered week 1. Otherwise, it is the last week of the previous
44038		year, and the next week is week 1. Both January 4th and the first Thursday of January
44039		are always in week 1. [<i>tm_year</i> , <i>tm_wday</i> , <i>tm_yday</i>]
44040	%w	Replaced by the weekday as a decimal number [0,6], with 0 representing Sunday.
44041		[<i>tm_wday</i>]
44042	%W	Replaced by the week number of the year as a decimal number [00,53]. The first
44043		Monday of January is the first day of week 1; days in the new year before this are in
44044		week 0. [<i>tm_year</i> , <i>tm_wday</i> , <i>tm_yday</i>]
44045	%x	Replaced by the locale's appropriate date representation. (See the Base Definitions
44046		volume of IEEE Std 1003.1-2001, <time.h>.)
44047	%X	Replaced by the locale's appropriate time representation. (See the Base Definitions
44048		volume of IEEE Std 1003.1-2001, <time.h>.)
44049	%y	Replaced by the last two digits of the year as a decimal number [00,99]. [<i>tm_year</i>]
44050	%Y	Replaced by the year as a decimal number (for example, 1997). [<i>tm_year</i>]
44051	%z	Replaced by the offset from UTC in the ISO 8601:2000 standard format (+hhmm or
44052		-hhmm), or by no characters if no timezone is determinable. For example, "-0430"
44053 CX		means 4 hours 30 minutes behind UTC (west of Greenwich). If <i>tm_isdst</i> is zero, the
44054		standard time offset is used. If <i>tm_isdst</i> is greater than zero, the daylight savings time
44055		offset is used. If <i>tm_isdst</i> is negative, no characters are returned. [<i>tm_isdst</i>]

44056	%Z	Replaced by the timezone name or abbreviation, or by no bytes if no timezone information exists. [<i>tm_isdst</i>]
44057		
44058	%%	Replaced by %.
44059		If a conversion specification does not correspond to any of the above, the behavior is undefined.
44060		Modified Conversion Specifiers
44061		Some conversion specifiers can be modified by the E or O modifier characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified conversion specifier. If the alternative format or specification does not exist for the current locale (see ERA in the Base Definitions volume of IEEE Std 1003.1-2001, Section 7.3.5, LC_TIME), the behavior shall be as if the unmodified conversion specification were used.
44062		
44063		
44064		
44065		
44066	%Ec	Replaced by the locale's alternative appropriate date and time representation.
44067	%EC	Replaced by the name of the base year (period) in the locale's alternative representation.
44068		
44069	%Ex	Replaced by the locale's alternative date representation.
44070	%EX	Replaced by the locale's alternative time representation.
44071	%Ey	Replaced by the offset from %EC (year only) in the locale's alternative representation.
44072	%EY	Replaced by the full alternative year representation.
44073	%Od	Replaced by the day of the month, using the locale's alternative numeric symbols, filled as needed with leading zeros if there is any alternative symbol for zero; otherwise, with leading spaces.
44074		
44075		
44076	%Oe	Replaced by the day of the month, using the locale's alternative numeric symbols, filled as needed with leading spaces.
44077		
44078	%OH	Replaced by the hour (24-hour clock) using the locale's alternative numeric symbols.
44079	%OI	Replaced by the hour (12-hour clock) using the locale's alternative numeric symbols.
44080	%Om	Replaced by the month using the locale's alternative numeric symbols.
44081	%OM	Replaced by the minutes using the locale's alternative numeric symbols.
44082	%OS	Replaced by the seconds using the locale's alternative numeric symbols.
44083	%Ou	Replaced by the weekday as a number in the locale's alternative representation (Monday=1).
44084		
44085	%OU	Replaced by the week number of the year (Sunday as the first day of the week, rules corresponding to %U) using the locale's alternative numeric symbols.
44086		
44087	%OV	Replaced by the week number of the year (Monday as the first day of the week, rules corresponding to %V) using the locale's alternative numeric symbols.
44088		
44089	%Ow	Replaced by the number of the weekday (Sunday=0) using the locale's alternative numeric symbols.
44090		
44091	%OW	Replaced by the week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.
44092		
44093	%Oy	Replaced by the year (offset from %C) using the locale's alternative numeric symbols.
44094	%g, %G, and %V	give values according to the ISO 8601:2000 standard week-based year. In this system, weeks begin on a Monday and week 1 of the year is the week that includes January 4th,
44095		

44096 which is also the week that includes the first Thursday of the year, and is also the first week that
 44097 contains at least four days in the year. If the first Monday of January is the 2nd, 3rd, or 4th, the
 44098 preceding days are part of the last week of the preceding year; thus, for Saturday 2nd January
 44099 1999, %G is replaced by 1998 and %V is replaced by 53. If December 29th, 30th, or 31st is a
 44100 Monday, it and any following days are part of week 1 of the following year. Thus, for Tuesday
 44101 30th December 1997, %G is replaced by 1998 and %V is replaced by 01.

44102 If a conversion specifier is not one of the above, the behavior is undefined.

44103 RETURN VALUE

44104 If the total number of resulting bytes including the terminating null byte is not more than
 44105 *maxsize*, *strptime()* shall return the number of bytes placed into the array pointed to by *s*, not
 44106 including the terminating null byte. Otherwise, 0 shall be returned and the contents of the array
 44107 are unspecified.

44108 ERRORS

44109 No errors are defined.

44110 EXAMPLES

44111 Getting a Localized Date String

44112 The following example first sets the locale to the user's default. The locale information will be
 44113 used in the *nl_langinfo()* and *strptime()* functions. The *nl_langinfo()* function returns the localized
 44114 date string which specifies how the date is laid out. The *strptime()* function takes this information
 44115 and, using the *tm* structure for values, places the date and time information into *datestring*.

```
44116 #include <time.h>
44117 #include <locale.h>
44118 #include <langinfo.h>
44119 ...
44120 struct tm *tm;
44121 char datestring[256];
44122 ...
44123 setlocale (LC_ALL, "");
44124 ...
44125 strptime (datestring, sizeof(datestring), nl_langinfo (D_T_FMT), tm);
44126 ...
```

44127 APPLICATION USAGE

44128 The range of values for %S is [00,60] rather than [00,59] to allow for the occasional leap second.

44129 Some of the conversion specifications are duplicates of others. They are included for
 44130 compatibility with *nl_cxtime()* and *nl_ascxtime()*, which were published in Issue 2.

44131 Applications should use %Y (4-digit years) in preference to %y (2-digit years).

44132 In the C locale, the E and O modifiers are ignored and the replacement strings for the following
 44133 specifiers are:

44134	%a	The first three characters of %A.
44135	%A	One of Sunday, Monday, ..., Saturday.
44136	%b	The first three characters of %B.
44137	%B	One of January, February, ..., December.
44138	%c	Equivalent to %a %b %e %T %Y.

44139	%p	One of AM or PM.
44140	%r	Equivalent to %I:%M:%S %p.
44141	%x	Equivalent to %m/%d/%Y.
44142	%X	Equivalent to %T.
44143	%Z	Implementation-defined.
44144	RATIONALE	
44145	None.	
44146	FUTURE DIRECTIONS	
44147	None.	
44148	SEE ALSO	
44149	<i>asctime()</i> , <i>clock()</i> , <i>ctime()</i> , <i>difftime()</i> , <i>getdate()</i> , <i>gmtime()</i> , <i>localtime()</i> , <i>mktime()</i> , <i>strptime()</i> , <i>time()</i> ,	
44150	<i>tzset()</i> , <i>utime()</i> , Base Definitions volume of IEEE Std 1003.1-2001, Section 7.3.5, LC_TIME,	
44151	<time.h>	
44152	CHANGE HISTORY	
44153	First released in Issue 3.	
44154	Issue 5	
44155	The description of %OV is changed to be consistent with %V and defines Monday as the first day	
44156	of the week.	
44157	The description of %Oy is clarified.	
44158	Issue 6	
44159	Extensions beyond the ISO C standard are marked.	
44160	The Open Group Corrigendum U033/8 is applied. The %V conversion specifier is changed from	
44161	“Otherwise, it is week 53 of the previous year, and the next week is week 1” to “Otherwise, it is	
44162	the last week of the previous year, and the next week is week 1”.	
44163	The following new requirements on POSIX implementations derive from alignment with the	
44164	Single UNIX Specification:	
44165	<ul style="list-style-type: none"> • The %C, %D, %e, %h, %n, %r, %R, %t, and %T conversion specifiers are added. 	
44166	<ul style="list-style-type: none"> • The modified conversion specifiers are added for consistency with the ISO POSIX-2 standard 	
44167	<i>date</i> utility.	
44168	The following changes are made for alignment with the ISO/IEC 9899: 1999 standard:	
44169	<ul style="list-style-type: none"> • The <i>strptime()</i> prototype is updated. 	
44170	<ul style="list-style-type: none"> • The DESCRIPTION is extensively revised. 	
44171	<ul style="list-style-type: none"> • The %z conversion specifier is added. 	
44172	A new example is added.	

44173 **NAME**

44174 strlen — get string length

44175 **SYNOPSIS**

44176 #include <string.h>

44177 size_t strlen(const char *s);

44178 **DESCRIPTION**

44179 cx The functionality described on this reference page is aligned with the ISO C standard. Any
 44180 conflict between the requirements described here and the ISO C standard is unintentional. This
 44181 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

44182 The *strlen()* function shall compute the number of bytes in the string to which *s* points, not
 44183 including the terminating null byte.

44184 **RETURN VALUE**

44185 The *strlen()* function shall return the length of *s*; no return value shall be reserved to indicate an
 44186 error.

44187 **ERRORS**

44188 No errors are defined.

44189 **EXAMPLES**44190 **Getting String Lengths**

44191 The following example sets the maximum length of *key* and *data* by using *strlen()* to get the
 44192 lengths of those strings.

```

44193     #include <string.h>
44194     ...
44195     struct element {
44196         char *key;
44197         char *data;
44198     };
44199     ...
44200     char *key, *data;
44201     int len;

44202     *keylength = *datalength = 0;
44203     ...
44204     if ((len = strlen(key)) > *keylength)
44205         *keylength = len;
44206     if ((len = strlen(data)) > *datalength)
44207         *datalength = len;
44208     ...
```

44209 **APPLICATION USAGE**

44210 None.

44211 **RATIONALE**

44212 None.

44213 **FUTURE DIRECTIONS**

44214 None.

44215 **SEE ALSO**44216 The Base Definitions volume of IEEE Std 1003.1-2001, <**string.h**>44217 **CHANGE HISTORY**

44218 First released in Issue 1. Derived from Issue 1 of the SVID.

44219 **Issue 5**44220 The RETURN VALUE section is updated to indicate that *strlen()* returns the length of *s*, and not
44221 *s* itself as was previously stated.

44222 **NAME**

44223 strncasecmp — case-insensitive string comparison

44224 **SYNOPSIS**

44225 XSI #include <strings.h>

44226 int strncasecmp(const char *s1, const char *s2, size_t n);

44227

44228 **DESCRIPTION**44229 Refer to *strcasecmp()*.

44230 **NAME**

44231 strncat — concatenate a string with part of another

44232 **SYNOPSIS**

44233 #include <string.h>

44234 char *strncat(char *restrict s1, const char *restrict s2, size_t n);

44235 **DESCRIPTION**

44236 cx The functionality described on this reference page is aligned with the ISO C standard. Any
44237 conflict between the requirements described here and the ISO C standard is unintentional. This
44238 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

44239 The *strncat()* function shall append not more than *n* bytes (a null byte and bytes that follow it
44240 are not appended) from the array pointed to by *s2* to the end of the string pointed to by *s1*. The
44241 initial byte of *s2* overwrites the null byte at the end of *s1*. A terminating null byte is always
44242 appended to the result. If copying takes place between objects that overlap, the behavior is
44243 undefined.

44244 **RETURN VALUE**44245 The *strncat()* function shall return *s1*; no return value shall be reserved to indicate an error.44246 **ERRORS**

44247 No errors are defined.

44248 **EXAMPLES**

44249 None.

44250 **APPLICATION USAGE**

44251 None.

44252 **RATIONALE**

44253 None.

44254 **FUTURE DIRECTIONS**

44255 None.

44256 **SEE ALSO**44257 *strcat()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**string.h**>44258 **CHANGE HISTORY**

44259 First released in Issue 1. Derived from Issue 1 of the SVID.

44260 **Issue 6**44261 The *strncat()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

44262 **NAME**

44263 strncmp — compare part of two strings

44264 **SYNOPSIS**

44265 #include <string.h>

44266 int strncmp(const char *s1, const char *s2, size_t n);

44267 **DESCRIPTION**

44268 cx The functionality described on this reference page is aligned with the ISO C standard. Any
44269 conflict between the requirements described here and the ISO C standard is unintentional. This
44270 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

44271 The *strncmp()* function shall compare not more than *n* bytes (bytes that follow a null byte are not
44272 compared) from the array pointed to by *s1* to the array pointed to by *s2*.

44273 The sign of a non-zero return value is determined by the sign of the difference between the
44274 values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the strings
44275 being compared.

44276 **RETURN VALUE**

44277 Upon successful completion, *strncmp()* shall return an integer greater than, equal to, or less than
44278 0, if the possibly null-terminated array pointed to by *s1* is greater than, equal to, or less than the
44279 possibly null-terminated array pointed to by *s2* respectively.

44280 **ERRORS**

44281 No errors are defined.

44282 **EXAMPLES**

44283 None.

44284 **APPLICATION USAGE**

44285 None.

44286 **RATIONALE**

44287 None.

44288 **FUTURE DIRECTIONS**

44289 None.

44290 **SEE ALSO**44291 *strcmp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**string.h**>44292 **CHANGE HISTORY**

44293 First released in Issue 1. Derived from Issue 1 of the SVID.

44294 **Issue 6**

44295 Extensions beyond the ISO C standard are marked.

44296 **NAME**

44297 strncpy — copy part of a string

44298 **SYNOPSIS**

44299 #include <string.h>

44300 char *strncpy(char *restrict s1, const char *restrict s2, size_t n);

44301 **DESCRIPTION**

44302 cx The functionality described on this reference page is aligned with the ISO C standard. Any
44303 conflict between the requirements described here and the ISO C standard is unintentional. This
44304 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

44305 The *strncpy()* function shall copy not more than *n* bytes (bytes that follow a null byte are not
44306 copied) from the array pointed to by *s2* to the array pointed to by *s1*. If copying takes place
44307 between objects that overlap, the behavior is undefined.

44308 If the array pointed to by *s2* is a string that is shorter than *n* bytes, null bytes shall be appended
44309 to the copy in the array pointed to by *s1*, until *n* bytes in all are written.

44310 **RETURN VALUE**44311 The *strncpy()* function shall return *s1*; no return value is reserved to indicate an error.44312 **ERRORS**

44313 No errors are defined.

44314 **EXAMPLES**

44315 None.

44316 **APPLICATION USAGE**

44317 Character movement is performed differently in different implementations. Thus, overlapping
44318 moves may yield surprises.

44319 If there is no null byte in the first *n* bytes of the array pointed to by *s2*, the result is not null-
44320 terminated.

44321 **RATIONALE**

44322 None.

44323 **FUTURE DIRECTIONS**

44324 None.

44325 **SEE ALSO**44326 *strcpy()*, the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>44327 **CHANGE HISTORY**

44328 First released in Issue 1. Derived from Issue 1 of the SVID.

44329 **Issue 6**44330 The *strncpy()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

44331 **NAME**

44332 strpbrk — scan a string for a byte

44333 **SYNOPSIS**

44334 #include <string.h>

44335 char *strpbrk(const char *s1, const char *s2);

44336 **DESCRIPTION**

44337 cx The functionality described on this reference page is aligned with the ISO C standard. Any
44338 conflict between the requirements described here and the ISO C standard is unintentional. This
44339 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

44340 The *strpbrk()* function shall locate the first occurrence in the string pointed to by *s1* of any byte
44341 from the string pointed to by *s2*.

44342 **RETURN VALUE**

44343 Upon successful completion, *strpbrk()* shall return a pointer to the byte or a null pointer if no
44344 byte from *s2* occurs in *s1*.

44345 **ERRORS**

44346 No errors are defined.

44347 **EXAMPLES**

44348 None.

44349 **APPLICATION USAGE**

44350 None.

44351 **RATIONALE**

44352 None.

44353 **FUTURE DIRECTIONS**

44354 None.

44355 **SEE ALSO**44356 *strchr()*, *strrchr()*, the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>44357 **CHANGE HISTORY**

44358 First released in Issue 1. Derived from Issue 1 of the SVID.

44359 NAME

44360 strptime — date and time conversion

44361 SYNOPSIS

44362 XSI `#include <time.h>`

```
44363 char *strptime(const char *restrict buf, const char *restrict format,
44364               struct tm *restrict tm);
44365
```

44366 DESCRIPTION

44367 The *strptime()* function shall convert the character string pointed to by *buf* to values which are
 44368 stored in the **tm** structure pointed to by *tm*, using the format specified by *format*.

44369 The *format* is composed of zero or more directives. Each directive is composed of one of the
 44370 following: one or more white-space characters (as specified by *isspace()*); an ordinary character
 44371 (neither '%' nor a white-space character); or a conversion specification. Each conversion
 44372 specification is composed of a '%' character followed by a conversion character which specifies
 44373 the replacement required. The application shall ensure that there is white-space or other non-
 44374 alphanumeric characters between any two conversion specifications. The following conversion
 44375 specifications are supported:

44376	%a	The day of the week, using the locale's weekday names; either the abbreviated or full name may be specified.
44377		
44378	%A	Equivalent to %a.
44379	%b	The month, using the locale's month names; either the abbreviated or full name may be specified.
44380		
44381	%B	Equivalent to %b.
44382	%c	Replaced by the locale's appropriate date and time representation.
44383	%C	The century number [00,99]; leading zeros are permitted but not required.
44384	%d	The day of the month [01,31]; leading zeros are permitted but not required.
44385	%D	The date as %m/%d/%y.
44386	%e	Equivalent to %d.
44387	%h	Equivalent to %b.
44388	%H	The hour (24-hour clock) [00,23]; leading zeros are permitted but not required.
44389	%I	The hour (12-hour clock) [01,12]; leading zeros are permitted but not required.
44390	%j	The day number of the year [001,366]; leading zeros are permitted but not required.
44391	%m	The month number [01,12]; leading zeros are permitted but not required.
44392	%M	The minute [00,59]; leading zeros are permitted but not required.
44393	%n	Any white space.
44394	%p	The locale's equivalent of a.m or p.m.
44395	%r	12-hour clock time using the AM/PM notation if t_fmt_ampm is not an empty string in the LC_TIME portion of the current locale; in the POSIX locale, this shall be equivalent to %I:%M:%S %p.
44396		
44397		
44398	%R	The time as %H:%M.

44399	%S	The seconds [00,60]; leading zeros are permitted but not required.
44400	%t	Any white space.
44401	%T	The time as %H:%M:%S.
44402	%U	The week number of the year (Sunday as the first day of the week) as a decimal number [00,53]; leading zeros are permitted but not required.
44403		
44404	%w	The weekday as a decimal number [0,6], with 0 representing Sunday; leading zeros are permitted but not required.
44405		
44406	%W	The week number of the year (Monday as the first day of the week) as a decimal number [00,53]; leading zeros are permitted but not required.
44407		
44408	%x	The date, using the locale's date format.
44409	%X	The time, using the locale's time format.
44410	%y	The year within century. When a century is not otherwise specified, values in the range [69,99] shall refer to years 1969 to 1999 inclusive, and values in the range [00,68] shall refer to years 2000 to 2068 inclusive; leading zeros shall be permitted but shall not be required.
44411		
44412		
44413		
44414	Note:	It is expected that in a future version of IEEE Std 1003.1-2001 the default century inferred from a 2-digit year will change. (This would apply to all commands accepting a 2-digit year as input.)
44415		
44416		
44417	%Y	The year, including the century (for example, 1988).
44418	%%	Replaced by %.

44419 **Modified Conversion Specifiers**

44420	Some conversion specifiers can be modified by the E and O modifier characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified conversion specifier. If the alternative format or specification does not exist in the current locale, the behavior shall be as if the unmodified conversion specification were used.	
44421		
44422		
44423		
44424	%Ec	The locale's alternative appropriate date and time representation.
44425	%EC	The name of the base year (period) in the locale's alternative representation.
44426	%Ex	The locale's alternative date representation.
44427	%EX	The locale's alternative time representation.
44428	%Ey	The offset from %EC (year only) in the locale's alternative representation.
44429	%EY	The full alternative year representation.
44430	%Od	The day of the month using the locale's alternative numeric symbols; leading zeros are permitted but not required.
44431		
44432	%Oe	Equivalent to %Od.
44433	%OH	The hour (24-hour clock) using the locale's alternative numeric symbols.
44434	%OI	The hour (12-hour clock) using the locale's alternative numeric symbols.
44435	%Om	The month using the locale's alternative numeric symbols.
44436	%OM	The minutes using the locale's alternative numeric symbols.

44437	%OS	The seconds using the locale's alternative numeric symbols.
44438	%OU	The week number of the year (Sunday as the first day of the week) using the locale's alternative numeric symbols.
44439		
44440	%Ow	The number of the weekday (Sunday=0) using the locale's alternative numeric symbols.
44441	%OW	The week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.
44442		
44443	%Oy	The year (offset from %C) using the locale's alternative numeric symbols.
44444		A conversion specification composed of white-space characters is executed by scanning input up to the first character that is not white-space (which remains unscanned), or until no more characters can be scanned.
44445		
44446		
44447		A conversion specification that is an ordinary character is executed by scanning the next character from the buffer. If the character scanned from the buffer differs from the one comprising the directive, the directive fails, and the differing and subsequent characters remain unscanned.
44448		
44449		
44450		
44451		A series of conversion specifications composed of %n, %t, white-space characters, or any combination is executed by scanning up to the first character that is not white space (which remains unscanned), or until no more characters can be scanned.
44452		
44453		
44454		Any other conversion specification is executed by scanning characters until a character matching the next directive is scanned, or until no more characters can be scanned. These characters, except the one matching the next directive, are then compared to the locale values associated with the conversion specifier. If a match is found, values for the appropriate tm structure members are set to values corresponding to the locale information. Case is ignored when matching items in <i>buf</i> such as month or weekday names. If no match is found, <i>strptime()</i> fails and no more characters are scanned.
44455		
44456		
44457		
44458		
44459		
44460		
44461	RETURN VALUE	
44462		Upon successful completion, <i>strptime()</i> shall return a pointer to the character following the last character parsed. Otherwise, a null pointer shall be returned.
44463		
44464	ERRORS	
44465		No errors are defined.
44466	EXAMPLES	
44467		None.
44468	APPLICATION USAGE	
44469		Several “equivalent to” formats and the special processing of white-space characters are provided in order to ease the use of identical <i>format</i> strings for <i>strftime()</i> and <i>strptime()</i> .
44470		
44471		Applications should use %Y (4-digit years) in preference to %y (2-digit years).
44472		It is unspecified whether multiple calls to <i>strptime()</i> using the same tm structure will update the current contents of the structure or overwrite all contents of the structure. Conforming applications should make a single call to <i>strptime()</i> with a format and all data needed to completely specify the date and time being converted.
44473		
44474		
44475		
44476	RATIONALE	
44477		None.

44478 **FUTURE DIRECTIONS**

44479 The *strptime()* function is expected to be mandatory in the next version of this volume of
44480 IEEE Std 1003.1-2001.

44481 **SEE ALSO**

44482 *scanf()*, *strptime()*, *time()*, the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>

44483 **CHANGE HISTORY**

44484 First released in Issue 4.

44485 **Issue 5**

44486 Moved from ENHANCED I18N to BASE.

44487 The [ENOSYS] error is removed.

44488 The exact meaning of the %y and %Oy specifiers is clarified in the DESCRIPTION.

44489 **Issue 6**

44490 The Open Group Corrigendum U033/5 is applied. The %r specifier description is reworded.

44491 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

44492 The **restrict** keyword is added to the *strptime()* prototype for alignment with the
44493 ISO/IEC 9899:1999 standard.

44494 The Open Group Corrigendum U047/2 is applied.

44495 The DESCRIPTION is updated to use the terms “conversion specifier” and “conversion
44496 specification” for consistency with *strptime()*.

44497 **NAME**

44498 strchr — string scanning operation

44499 **SYNOPSIS**

44500 #include <string.h>

44501 char *strchr(const char *s, int c);

44502 **DESCRIPTION**

44503 cx The functionality described on this reference page is aligned with the ISO C standard. Any
44504 conflict between the requirements described here and the ISO C standard is unintentional. This
44505 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

44506 The *strchr()* function shall locate the last occurrence of *c* (converted to a **char**) in the string
44507 pointed to by *s*. The terminating null byte is considered to be part of the string.

44508 **RETURN VALUE**

44509 Upon successful completion, *strchr()* shall return a pointer to the byte or a null pointer if *c* does
44510 not occur in the string.

44511 **ERRORS**

44512 No errors are defined.

44513 **EXAMPLES**44514 **Finding the Base Name of a File**

44515 The following example uses *strchr()* to get a pointer to the base name of a file. The *strchr()*
44516 function searches backwards through the name of the file to find the last '/' character in *name*.
44517 This pointer (plus one) will point to the base name of the file.

```
44518     #include <string.h>
44519     ...
44520     const char *name;
44521     char *basename;
44522     ...
44523     basename = strchr(name, '/') + 1;
44524     ...
```

44525 **APPLICATION USAGE**

44526 None.

44527 **RATIONALE**

44528 None.

44529 **FUTURE DIRECTIONS**

44530 None.

44531 **SEE ALSO**44532 *strchr()*, the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>44533 **CHANGE HISTORY**

44534 First released in Issue 1. Derived from Issue 1 of the SVID.

44535 **NAME**

44536 strspn — get length of a substring

44537 **SYNOPSIS**

44538 #include <string.h>

44539 size_t strspn(const char *s1, const char *s2);

44540 **DESCRIPTION**

44541 cx The functionality described on this reference page is aligned with the ISO C standard. Any
44542 conflict between the requirements described here and the ISO C standard is unintentional. This
44543 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

44544 The *strspn()* function shall compute the length (in bytes) of the maximum initial segment of the
44545 string pointed to by *s1* which consists entirely of bytes from the string pointed to by *s2*.

44546 **RETURN VALUE**

44547 The *strspn()* function shall return the length of *s1*; no return value is reserved to indicate an
44548 error.

44549 **ERRORS**

44550 No errors are defined.

44551 **EXAMPLES**

44552 None.

44553 **APPLICATION USAGE**

44554 None.

44555 **RATIONALE**

44556 None.

44557 **FUTURE DIRECTIONS**

44558 None.

44559 **SEE ALSO**44560 *strcspn()*, the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>44561 **CHANGE HISTORY**

44562 First released in Issue 1. Derived from Issue 1 of the SVID.

44563 **Issue 5**

44564 The RETURN VALUE section is updated to indicate that *strspn()* returns the length of *s*, and not
44565 *s* itself as was previously stated.

44566 **NAME**44567 **strstr** — find a substring44568 **SYNOPSIS**44569 `#include <string.h>`44570 `char *strstr(const char *s1, const char *s2);`44571 **DESCRIPTION**

44572 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
44573 conflict between the requirements described here and the ISO C standard is unintentional. This
44574 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

44575 The *strstr()* function shall locate the first occurrence in the string pointed to by *s1* of the
44576 sequence of bytes (excluding the terminating null byte) in the string pointed to by *s2*.

44577 **RETURN VALUE**

44578 Upon successful completion, *strstr()* shall return a pointer to the located string or a null pointer
44579 if the string is not found.

44580 If *s2* points to a string with zero length, the function shall return *s1*.

44581 **ERRORS**

44582 No errors are defined.

44583 **EXAMPLES**

44584 None.

44585 **APPLICATION USAGE**

44586 None.

44587 **RATIONALE**

44588 None.

44589 **FUTURE DIRECTIONS**

44590 None.

44591 **SEE ALSO**

44592 *strchr()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<string.h>**

44593 **CHANGE HISTORY**

44594 First released in Issue 3. Included for alignment with the ANSI C standard.

44595 NAME

44596 strtod, strtodf, strtold — convert a string to a double-precision number

44597 SYNOPSIS

44598 #include <stdlib.h>

44599 double strtod(const char *restrict nptr, char **restrict endptr);

44600 float strtodf(const char *restrict nptr, char **restrict endptr);

44601 long double strtold(const char *restrict nptr, char **restrict endptr);

44602 DESCRIPTION

44603 cx The functionality described on this reference page is aligned with the ISO C standard. Any
 44604 conflict between the requirements described here and the ISO C standard is unintentional. This
 44605 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

44606 These functions shall convert the initial portion of the string pointed to by *nptr* to **double**, **float**,
 44607 and **long double** representation, respectively. First, they decompose the input string into three
 44608 parts:

- 44609 1. An initial, possibly empty, sequence of white-space characters (as specified by *isspace()*)
- 44610 2. A subject sequence interpreted as a floating-point constant or representing infinity or NaN
- 44611 3. A final string of one or more unrecognized characters, including the terminating null byte
- 44612 of the input string

44613 Then they shall attempt to convert the subject sequence to a floating-point number, and return
 44614 the result.

44615 The expected form of the subject sequence is an optional plus or minus sign, then one of the
 44616 following:

- 44617 • A non-empty sequence of decimal digits optionally containing a radix character, then an
 44618 optional exponent part
- 44619 • A 0x or 0X, then a non-empty sequence of hexadecimal digits optionally containing a radix
 44620 character, then an optional binary exponent part
- 44621 • One of INF or INFINITY, ignoring case
- 44622 • One of NAN or NAN(*n-char-sequence_{opt}*), ignoring case in the NAN part, where:

```

44623 n-char-sequence:
44624     digit
44625     nondigit
44626     n-char-sequence digit
44627     n-char-sequence nondigit
  
```

44628 The subject sequence is defined as the longest initial subsequence of the input string, starting
 44629 with the first non-white-space character, that is of the expected form. The subject sequence
 44630 contains no characters if the input string is not of the expected form.

44631 If the subject sequence has the expected form for a floating-point number, the sequence of
 44632 characters starting with the first digit or the decimal-point character (whichever occurs first)
 44633 shall be interpreted as a floating constant of the C language, except that the radix character shall
 44634 be used in place of a period, and that if neither an exponent part nor a radix character appears in
 44635 a decimal floating-point number, or if a binary exponent part does not appear in a hexadecimal
 44636 floating-point number, an exponent part of the appropriate type with value zero is assumed to
 44637 follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence
 44638 shall be interpreted as negated. A character sequence INF or INFINITY shall be interpreted as an

44639 infinity, if representable in the return type, else as if it were a floating constant that is too large
 44640 for the range of the return type. A character sequence NAN or NAN(*n-char-sequence_{opt}*) shall be
 44641 interpreted as a quiet NaN, if supported in the return type, else as if it were a subject sequence
 44642 part that does not have the expected form; the meaning of the *n-char* sequences is
 44643 implementation-defined. A pointer to the final string is stored in the object pointed to by *endptr*,
 44644 provided that *endptr* is not a null pointer.

44645 If the subject sequence has the hexadecimal form and FLT_RADIX is a power of 2, the value
 44646 resulting from the conversion is correctly rounded.

44647 CX The radix character is defined in the program's locale (category *LC_NUMERIC*). In the POSIX
 44648 locale, or in a locale where the radix character is not defined, the radix character shall default to a
 44649 period ('.').

44650 CX In other than the C or POSIX locales, other implementation-defined subject sequences may be
 44651 accepted.

44652 If the subject sequence is empty or does not have the expected form, no conversion shall be
 44653 performed; the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not
 44654 a null pointer.

44655 CX The *strtod()* function shall not change the setting of *errno* if successful.

44656 Since 0 is returned on error and is also a valid return on success, an application wishing to check
 44657 for error situations should set *errno* to 0, then call *strtod()*, *strtof()*, or *strtold()*, then check *errno*.

44658 RETURN VALUE

44659 Upon successful completion, these functions shall return the converted value. If no conversion
 44660 could be performed, 0 shall be returned, and *errno* may be set to [EINVAL].

44661 If the correct value is outside the range of representable values, HUGE_VAL, HUGE_VALF, or
 44662 HUGE_VALL shall be returned (according to the sign of the value), and *errno* shall be set to
 44663 [ERANGE].

44664 If the correct value would cause an underflow, a value whose magnitude is no greater than the
 44665 smallest normalized positive number in the return type shall be returned and *errno* set to
 44666 [ERANGE].

44667 ERRORS

44668 These functions shall fail if:

44669 CX [ERANGE] The value to be returned would cause overflow or underflow.

44670 These functions may fail if:

44671 CX [EINVAL] No conversion could be performed.

44672 EXAMPLES

44673 None.

44674 APPLICATION USAGE

44675 If the subject sequence has the hexadecimal form and FLT_RADIX is not a power of 2, and the
 44676 result is not exactly representable, the result should be one of the two numbers in the
 44677 appropriate internal format that are adjacent to the hexadecimal floating source value, with the
 44678 extra stipulation that the error should have a correct sign for the current rounding direction.

44679 If the subject sequence has the decimal form and at most DECIMAL_DIG (defined in <float.h>)
 44680 significant digits, the result should be correctly rounded. If the subject sequence *D* has the
 44681 decimal form and more than DECIMAL_DIG significant digits, consider the two bounding,
 44682 adjacent decimal strings *L* and *U*, both having DECIMAL_DIG significant digits, such that the

values of L , D , and U satisfy $L \leq D \leq U$. The result should be one of the (equal or adjacent) values that would be obtained by correctly rounding L and U according to the current rounding direction, with the extra stipulation that the error with respect to D should have a correct sign for the current rounding direction.

The changes to *strtod()* introduced by the ISO/IEC 9899:1999 standard can alter the behavior of well-formed applications complying with the ISO/IEC 9899:1990 standard and thus earlier versions of the base documents. One such example would be:

```
int
what_kind_of_number (char *s)
{
    char *endp;
    double d;
    long l;

    d = strtod(s, &endp);
    if (s != endp && *endp == '\\0')
        printf("It's a float with value %g\\n", d);
    else
    {
        l = strtol(s, &endp, 0);
        if (s != endp && *endp == '\\0')
            printf("It's an integer with value %ld\\n", l);
        else
            return 1;
    }
    return 0;
}
```

If the function is called with:

```
what_kind_of_number ("0x10")
```

an ISO/IEC 9899:1990 standard-compliant library will result in the function printing:

```
It's an integer with value 16
```

With the ISO/IEC 9899:1999 standard, the result is:

```
It's a float with value 16
```

The change in behavior is due to the inclusion of floating-point numbers in hexadecimal notation without requiring that either a decimal point or the binary exponent be present.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isspace(), *localeconv()*, *scanf()*, *setlocale()*, *strtol()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, **<float.h>**, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

44726 **Issue 5**

44727 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

44728 **Issue 6**

44729 Extensions beyond the ISO C standard are marked.

44730 The following new requirements on POSIX implementations derive from alignment with the
44731 Single UNIX Specification:

- 44732 • In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
44733 added if no conversion could be performed.

44734 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 44735 • The *strtod()* function is updated.
- 44736 • The *strtodf()* and *strtold()* functions are added.
- 44737 • The DESCRIPTION is extensively revised.

44738 ISO/IEC 9899:1999 standard, Technical Corrigendum No. 1 is incorporated.

44739 **NAME**

44740 strtoimax, strtoumax — convert string to integer type

44741 **SYNOPSIS**

44742 #include <inttypes.h>

44743 intmax_t strtoimax(const char *restrict nptr, char **restrict endptr,
44744 int base);44745 uintmax_t strtoumax(const char *restrict nptr, char **restrict endptr,
44746 int base);44747 **DESCRIPTION**44748 CX The functionality described on this reference page is aligned with the ISO C standard. Any
44749 conflict between the requirements described here and the ISO C standard is unintentional. This
44750 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.44751 These functions shall be equivalent to the *strtol()*, *strtoll()*, *strtoul()*, and *strtoull()* functions,
44752 except that the initial portion of the string shall be converted to **intmax_t** and **uintmax_t**
44753 representation, respectively.44754 **RETURN VALUE**

44755 These functions shall return the converted value, if any.

44756 If no conversion could be performed, zero shall be returned.

44757 If the correct value is outside the range of representable values, {INTMAX_MAX},
44758 {INTMAX_MIN}, or {UINTMAX_MAX} shall be returned (according to the return type and sign
44759 of the value, if any), and *errno* shall be set to [ERANGE].44760 **ERRORS**

44761 These functions shall fail if:

44762 [ERANGE] The value to be returned is not representable.

44763 These functions may fail if:

44764 [EINVAL] The value of *base* is not supported.44765 **EXAMPLES**

44766 None.

44767 **APPLICATION USAGE**

44768 None.

44769 **RATIONALE**

44770 None.

44771 **FUTURE DIRECTIONS**

44772 None.

44773 **SEE ALSO**44774 *strtol()*, *strtoul()*, the Base Definitions volume of IEEE Std 1003.1-2001, <inttypes.h>44775 **CHANGE HISTORY**

44776 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

44777 NAME

44778 strtok, strtok_r — split string into tokens

44779 SYNOPSIS

44780 #include <string.h>

44781 char *strtok(char *restrict s1, const char *restrict s2);

44782 TSF char *strtok_r(char *restrict s, const char *restrict sep,

44783 char **restrict lasts);

44784

44785 DESCRIPTION

44786 CX For *strtok()*: The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

44789 A sequence of calls to *strtok()* breaks the string pointed to by *s1* into a sequence of tokens, each of which is delimited by a byte from the string pointed to by *s2*. The first call in the sequence has *s1* as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by *s2* may be different from call to call.

44793 The first call in the sequence searches the string pointed to by *s1* for the first byte that is *not* contained in the current separator string pointed to by *s2*. If no such byte is found, then there are no tokens in the string pointed to by *s1* and *strtok()* shall return a null pointer. If such a byte is found, it is the start of the first token.

44797 The *strtok()* function then searches from there for a byte that *is* contained in the current separator string. If no such byte is found, the current token extends to the end of the string pointed to by *s1*, and subsequent searches for a token shall return a null pointer. If such a byte is found, it is overwritten by a null byte, which terminates the current token. The *strtok()* function saves a pointer to the following byte, from which the next search for a token shall start.

44802 Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

44804 The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-2001 calls *strtok()*.

44806 CX The *strtok()* function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

44808 TSF The *strtok_r()* function considers the null-terminated string *s* as a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *sep*. The argument *lasts* points to a user-provided pointer which points to stored information necessary for *strtok_r()* to continue scanning the same string.

44812 In the first call to *strtok_r()*, *s* points to a null-terminated string, *sep* to a null-terminated string of separator characters, and the value pointed to by *lasts* is ignored. The *strtok_r()* function shall return a pointer to the first character of the first token, write a null character into *s* immediately following the returned token, and update the pointer to which *lasts* points.

44816 In subsequent calls, *s* is a NULL pointer and *lasts* shall be unchanged from the previous call so that subsequent calls shall move through the string *s*, returning successive tokens until no tokens remain. The separator string *sep* may be different from call to call. When no token remains in *s*, a NULL pointer shall be returned.

44820 **RETURN VALUE**

44821 Upon successful completion, *strtok()* shall return a pointer to the first byte of a token. Otherwise,
 44822 if there is no token, *strtok()* shall return a null pointer.

44823 TSF The *strtok_r()* function shall return a pointer to the token found, or a NULL pointer when no
 44824 token is found.

44825 **ERRORS**

44826 No errors are defined.

44827 **EXAMPLES**44828 **Searching for Word Separators**

44829 The following example searches for tokens separated by <space>s.

```
44830 #include <string.h>
44831 ...
44832 char *token;
44833 char *line = "LINE TO BE SEPARATED";
44834 char *search = " ";

44835 /* Token will point to "LINE". */
44836 token = strtok(line, search);

44837 /* Token will point to "TO". */
44838 token = strtok(NULL, search);
```

44839 **Breaking a Line**

44840 The following example uses *strtok()* to break a line into two character strings separated by any
 44841 combination of <space>s, <tab>s, or <newline>s.

```
44842 #include <string.h>
44843 ...
44844 struct element {
44845     char *key;
44846     char *data;
44847 };
44848 ...
44849 char line[LINE_MAX];
44850 char *key, *data;
44851 ...
44852 key = strtok(line, " \n");
44853 data = strtok(NULL, " \n");
44854 ...
```

44855 **APPLICATION USAGE**

44856 The *strtok_r()* function is thread-safe and stores its state in a user-supplied buffer instead of
 44857 possibly using a static data area that may be overwritten by an unrelated call from another
 44858 thread.

44859 **RATIONALE**

44860 The *strtok()* function searches for a separator string within a larger string. It returns a pointer to
 44861 the last substring between separator strings. This function uses static storage to keep track of
 44862 the current string position between calls. The new function, *strtok_r()*, takes an additional
 44863 argument, *lasts*, to keep track of the current position in the string.

44864 **FUTURE DIRECTIONS**

44865 None.

44866 **SEE ALSO**44867 The Base Definitions volume of IEEE Std 1003.1-2001, <**string.h**>44868 **CHANGE HISTORY**

44869 First released in Issue 1. Derived from Issue 1 of the SVID.

44870 **Issue 5**44871 The *strtok_r()* function is included for alignment with the POSIX Threads Extension.44872 A note indicating that the *strtok()* function need not be reentrant is added to the DESCRIPTION.44873 **Issue 6**

44874 Extensions beyond the ISO C standard are marked.

44875 The *strtok_r()* function is marked as part of the Thread-Safe Functions option.

44876 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

44877 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
44878 its avoidance of possibly using a static data area.44879 The **restrict** keyword is added to the *strtok()* and *strtok_r()* prototypes for alignment with the
44880 ISO/IEC 9899:1999 standard.

44881 NAME

44882 strtol, strtoll — convert a string to a long integer

44883 SYNOPSIS

44884 #include <stdlib.h>

44885 long strtol(const char *restrict *str*, char **restrict *endptr*, int *base*);44886 long long strtoll(const char *restrict *str*, char **restrict *endptr*,44887 int *base*)

44888 DESCRIPTION

44889 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 44890 conflict between the requirements described here and the ISO C standard is unintentional. This
 44891 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

44892 These functions shall convert the initial portion of the string pointed to by *str* to a type **long** and
 44893 **long long** representation, respectively. First, they decompose the input string into three parts:

- 44894 1. An initial, possibly empty, sequence of white-space characters (as specified by *isspace()*)
- 44895 2. A subject sequence interpreted as an integer represented in some radix determined by the
 44896 value of *base*
- 44897 3. A final string of one or more unrecognized characters, including the terminating null byte
 44898 of the input string.

44899 Then they shall attempt to convert the subject sequence to an integer, and return the result.

44900 If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant,
 44901 octal constant, or hexadecimal constant, any of which may be preceded by a '+' or '-' sign. A
 44902 decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An
 44903 octal constant consists of the prefix '0' optionally followed by a sequence of the digits '0' to
 44904 '7' only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the
 44905 decimal digits and letters 'a' (or 'A') to 'f' (or 'F') with values 10 to 15 respectively.

44906 If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence
 44907 of letters and digits representing an integer with the radix specified by *base*, optionally preceded
 44908 by a '+' or '-' sign. The letters from 'a' (or 'A') to 'z' (or 'Z') inclusive are ascribed the
 44909 values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the
 44910 value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and
 44911 digits, following the sign if present.

44912 The subject sequence is defined as the longest initial subsequence of the input string, starting
 44913 with the first non-white-space character that is of the expected form. The subject sequence shall
 44914 contain no characters if the input string is empty or consists entirely of white-space characters,
 44915 or if the first non-white-space character is other than a sign or a permissible letter or digit.

44916 If the subject sequence has the expected form and the value of *base* is 0, the sequence of
 44917 characters starting with the first digit shall be interpreted as an integer constant. If the subject
 44918 sequence has the expected form and the value of *base* is between 2 and 36, it shall be used as the
 44919 base for conversion, ascribing to each letter its value as given above. If the subject sequence
 44920 begins with a minus sign, the value resulting from the conversion shall be negated. A pointer to
 44921 the final string shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null
 44922 pointer.

44923 CX In other than the C or POSIX locales, other implementation-defined subject sequences may be
 44924 accepted.

44925 If the subject sequence is empty or does not have the expected form, no conversion is performed;
 44926 the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null
 44927 pointer.

44928 CX The *strtol()* function shall not change the setting of *errno* if successful.

44929 Since 0, {LONG_MIN} or {LLONG_MIN}, and {LONG_MAX} or {LLONG_MAX} are returned on
 44930 error and are also valid returns on success, an application wishing to check for error situations
 44931 should set *errno* to 0, then call *strtol()* or *strtoll()*, then check *errno*.

44932 RETURN VALUE

44933 Upon successful completion, these functions shall return the converted value, if any. If no
 44934 CX conversion could be performed, 0 shall be returned and *errno* may be set to [EINVAL].

44935 If the correct value is outside the range of representable values, {LONG_MIN}, {LONG_MAX},
 44936 {LLONG_MIN}, or {LLONG_MAX} shall be returned (according to the sign of the value), and
 44937 *errno* set to [ERANGE].

44938 ERRORS

44939 These functions shall fail if:

44940 [ERANGE] The value to be returned is not representable.

44941 These functions may fail if:

44942 CX [EINVAL] The value of *base* is not supported.

44943 EXAMPLES

44944 None.

44945 APPLICATION USAGE

44946 None.

44947 RATIONALE

44948 None.

44949 FUTURE DIRECTIONS

44950 None.

44951 SEE ALSO

44952 *isalpha()*, *scanf()*, *strtod()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

44953 CHANGE HISTORY

44954 First released in Issue 1. Derived from Issue 1 of the SVID.

44955 Issue 5

44956 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

44957 Issue 6

44958 Extensions beyond the ISO C standard are marked.

44959 The following new requirements on POSIX implementations derive from alignment with the
 44960 Single UNIX Specification:

- 44961 • In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
- 44962 added if no conversion could be performed.

44963 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 44964 • The *strtol()* prototype is updated.
- 44965 • The *strtoll()* function is added.

44966 **NAME**44967 **strtold** — convert a string to a double-precision number44968 **SYNOPSIS**44969 `#include <stdlib.h>`44970 `long double strtold(const char *restrict nptr, char **restrict endptr);`44971 **DESCRIPTION**44972 Refer to *strtod()*.

44973 **NAME**44974 **strtoll** — convert a string to a long integer44975 **SYNOPSIS**44976 `#include <stdlib.h>`44977 `long long strtoll(const char *restrict str, char **restrict endptr,`
44978 `int base);`44979 **DESCRIPTION**44980 Refer to *strtol()*.

44981 NAME

44982 strtoul, strtoull — convert a string to an unsigned long

44983 SYNOPSIS

44984 #include <stdlib.h>

```

44985 unsigned long strtoul(const char *restrict str,
44986                      char **restrict endptr, int base);
44987 unsigned long long strtoull(const char *restrict str,
44988                             char **restrict endptr, int base);

```

44989 DESCRIPTION

44990 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 44991 conflict between the requirements described here and the ISO C standard is unintentional. This
 44992 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

44993 These functions shall convert the initial portion of the string pointed to by *str* to a type **unsigned**
 44994 **long** and **unsigned long long** representation, respectively. First, they decompose the input
 44995 string into three parts:

- 44996 1. An initial, possibly empty, sequence of white-space characters (as specified by *isspace()*)
- 44997 2. A subject sequence interpreted as an integer represented in some radix determined by the
 44998 value of *base*
- 44999 3. A final string of one or more unrecognized characters, including the terminating null byte
 45000 of the input string

45001 Then they shall attempt to convert the subject sequence to an unsigned integer, and return the
 45002 result.

45003 If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant,
 45004 octal constant, or hexadecimal constant, any of which may be preceded by a '+' or '-' sign. A
 45005 decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An
 45006 octal constant consists of the prefix '0' optionally followed by a sequence of the digits '0' to
 45007 '7' only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the
 45008 decimal digits and letters 'a' (or 'A') to 'f' (or 'F') with values 10 to 15 respectively.

45009 If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence
 45010 of letters and digits representing an integer with the radix specified by *base*, optionally preceded
 45011 by a '+' or '-' sign. The letters from 'a' (or 'A') to 'z' (or 'Z') inclusive are ascribed the
 45012 values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the
 45013 value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and
 45014 digits, following the sign if present.

45015 The subject sequence is defined as the longest initial subsequence of the input string, starting
 45016 with the first non-white-space character that is of the expected form. The subject sequence shall
 45017 contain no characters if the input string is empty or consists entirely of white-space characters,
 45018 or if the first non-white-space character is other than a sign or a permissible letter or digit.

45019 If the subject sequence has the expected form and the value of *base* is 0, the sequence of
 45020 characters starting with the first digit shall be interpreted as an integer constant. If the subject
 45021 sequence has the expected form and the value of *base* is between 2 and 36, it shall be used as the
 45022 base for conversion, ascribing to each letter its value as given above. If the subject sequence
 45023 begins with a minus sign, the value resulting from the conversion shall be negated. A pointer to
 45024 the final string shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null
 45025 pointer.

45026 CX In other than the C or POSIX locales, other implementation-defined subject sequences may be
45027 accepted.

45028 If the subject sequence is empty or does not have the expected form, no conversion shall be
45029 performed; the value of *str* shall be stored in the object pointed to by *endptr*, provided that *endptr*
45030 is not a null pointer.

45031 CX The *strtol()* function shall not change the setting of *errno* if successful.

45032 Since 0, {ULONG_MAX}, and {ULLONG_MAX} are returned on error and are also valid returns
45033 on success, an application wishing to check for error situations should set *errno* to 0, then call
45034 *strtol()* or *strtoll()*, then check *errno*.

45035 RETURN VALUE

45036 Upon successful completion, these functions shall return the converted value, if any. If no
45037 CX conversion could be performed, 0 shall be returned and *errno* may be set to [EINVAL]. If the
45038 correct value is outside the range of representable values, {ULONG_MAX} or {ULLONG_MAX}
45039 shall be returned and *errno* set to [ERANGE].

45040 ERRORS

45041 These functions shall fail if:

45042 CX [EINVAL] The value of *base* is not supported.

45043 [ERANGE] The value to be returned is not representable.

45044 These functions may fail if:

45045 CX [EINVAL] No conversion could be performed.

45046 EXAMPLES

45047 None.

45048 APPLICATION USAGE

45049 None.

45050 RATIONALE

45051 None.

45052 FUTURE DIRECTIONS

45053 None.

45054 SEE ALSO

45055 *isalpha()*, *scanf()*, *strtod()*, *strtol()*, the Base Definitions volume of IEEE Std 1003.1-2001,
45056 <stdlib.h>

45057 CHANGE HISTORY

45058 First released in Issue 4. Derived from the ANSI C standard.

45059 Issue 5

45060 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

45061 Issue 6

45062 Extensions beyond the ISO C standard are marked.

45063 The following new requirements on POSIX implementations derive from alignment with the
45064 Single UNIX Specification:

- 45065 • The [EINVAL] error condition is added for when the value of *base* is not supported.

45066 In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
45067 added if no conversion could be performed.

- 45068 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:
- 45069 • The *strtoul()* prototype is updated.
- 45070 • The *strtoull()* function is added.

45071 **NAME**45072 **strtoumax** — convert a string to an integer type45073 **SYNOPSIS**45074 `#include <inttypes.h>`45075 `uintmax_t strtoumax(const char *restrict nptr, char **restrict endptr,`
45076 `int base);`45077 **DESCRIPTION**45078 Refer to *strtoimax()*.

45079 **NAME**45080 `strxfrm` — string transformation45081 **SYNOPSIS**45082 `#include <string.h>`45083 `size_t strxfrm(char *restrict s1, const char *restrict s2, size_t n);`45084 **DESCRIPTION**

45085 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 45086 conflict between the requirements described here and the ISO C standard is unintentional. This
 45087 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

45088 The `strxfrm()` function shall transform the string pointed to by `s2` and place the resulting string
 45089 into the array pointed to by `s1`. The transformation is such that if `strcmp()` is applied to two
 45090 transformed strings, it shall return a value greater than, equal to, or less than 0, corresponding to
 45091 the result of `strcoll()` applied to the same two original strings. No more than `n` bytes are placed
 45092 into the resulting array pointed to by `s1`, including the terminating null byte. If `n` is 0, `s1` is
 45093 permitted to be a null pointer. If copying takes place between objects that overlap, the behavior
 45094 is undefined.

45095 CX The `strxfrm()` function shall not change the setting of `errno` if successful.

45096 Since no return value is reserved to indicate an error, an application wishing to check for error
 45097 situations should set `errno` to 0, then call `strxfrm()`, then check `errno`.

45098 **RETURN VALUE**

45099 Upon successful completion, `strxfrm()` shall return the length of the transformed string (not
 45100 including the terminating null byte). If the value returned is `n` or more, the contents of the array
 45101 pointed to by `s1` are unspecified.

45102 CX On error, `strxfrm()` may set `errno` but no return value is reserved to indicate an error.

45103 **ERRORS**

45104 The `strxfrm()` function may fail if:

45105 CX [EINVAL] The string pointed to by the `s2` argument contains characters outside the
 45106 domain of the collating sequence.

45107 **EXAMPLES**

45108 None.

45109 **APPLICATION USAGE**

45110 The transformation function is such that two transformed strings can be ordered by `strcmp()` as
 45111 appropriate to collating sequence information in the program's locale (category `LC_COLLATE`).

45112 The fact that when `n` is 0 `s1` is permitted to be a null pointer is useful to determine the size of the
 45113 `s1` array prior to making the transformation.

45114 **RATIONALE**

45115 None.

45116 **FUTURE DIRECTIONS**

45117 None.

45118 **SEE ALSO**

45119 `strcmp()`, `strcoll()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<string.h>`

45120 **CHANGE HISTORY**

45121 First released in Issue 3. Included for alignment with the ISO C standard.

45122 **Issue 5**

45123 The DESCRIPTION is updated to indicate that *errno* does not change if the function is
45124 successful.

45125 **Issue 6**

45126 Extensions beyond the ISO C standard are marked.

45127 The following new requirements on POSIX implementations derive from alignment with the
45128 Single UNIX Specification:

- 45129 • In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
45130 added if no conversion could be performed.

45131 The *strxfrm()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

45132 **NAME**

45133 swab — swap bytes

45134 **SYNOPSIS**

45135 XSI #include <unistd.h>

45136 void swab(const void *restrict *src*, void *restrict *dest*,
45137 ssize_t *nbytes*);

45138

45139 **DESCRIPTION**

45140 The *swab()* function shall copy *nbytes* bytes, which are pointed to by *src*, to the object pointed to
45141 by *dest*, exchanging adjacent bytes. The *nbytes* argument should be even. If *nbytes* is odd, *swab()*
45142 copies and exchanges *nbytes*–1 bytes and the disposition of the last byte is unspecified. If
45143 copying takes place between objects that overlap, the behavior is undefined. If *nbytes* is
45144 negative, *swab()* does nothing.

45145 **RETURN VALUE**

45146 None.

45147 **ERRORS**

45148 No errors are defined.

45149 **EXAMPLES**

45150 None.

45151 **APPLICATION USAGE**

45152 None.

45153 **RATIONALE**

45154 None.

45155 **FUTURE DIRECTIONS**

45156 None.

45157 **SEE ALSO**

45158 The Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

45159 **CHANGE HISTORY**

45160 First released in Issue 1. Derived from Issue 1 of the SVID.

45161 **Issue 6**

45162 The **restrict** keyword is added to the *swab()* prototype for alignment with the
45163 ISO/IEC 9899:1999 standard.

45164 NAME

45165 swapcontext — swap user context

45166 SYNOPSIS

45167 XSI `#include <ucontext.h>`

```
45168 int swapcontext(ucontext_t *restrict oucp,  
45169                const ucontext_t *restrict ucp);
```

45170

45171 DESCRIPTION

45172 Refer to *makecontext()*.

45173 **NAME**

45174 swprintf — print formatted wide-character output

45175 **SYNOPSIS**

45176 #include <stdio.h>

45177 #include <wchar.h>

45178 int swprintf(wchar_t *restrict *ws*, size_t *n*,45179 const wchar_t *restrict *format*, ...);45180 **DESCRIPTION**45181 Refer to *fwprintf()*.

45182 **NAME**

45183 swscanf — convert formatted wide-character input

45184 **SYNOPSIS**

45185 #include <stdio.h>

45186 #include <wchar.h>

45187 int swscanf(const wchar_t *restrict *ws*,45188 const wchar_t *restrict *format*, ...);45189 **DESCRIPTION**45190 Refer to *fwscanf()*.

45191 **NAME**45192 `symlink` — make a symbolic link to a file45193 **SYNOPSIS**45194 `#include <unistd.h>`45195 `int symlink(const char *path1, const char *path2);`45196 **DESCRIPTION**

45197 The `symlink()` function shall create a symbolic link called *path2* that contains the string pointed
 45198 to by *path1* (*path2* is the name of the symbolic link created, *path1* is the string contained in the
 45199 symbolic link).

45200 The string pointed to by *path1* shall be treated only as a character string and shall not be
 45201 validated as a pathname.

45202 If the `symlink()` function fails for any reason other than [EIO], any file named by *path2* shall be
 45203 unaffected.

45204 **RETURN VALUE**

45205 Upon successful completion, `symlink()` shall return 0; otherwise, it shall return -1 and set *errno* to
 45206 indicate the error.

45207 **ERRORS**45208 The `symlink()` function shall fail if:

45209 [EACCES] Write permission is denied in the directory where the symbolic link is being
 45210 created, or search permission is denied for a component of the path prefix of
 45211 *path2*.

45212 [EEXIST] The *path2* argument names an existing file or symbolic link.

45213 [EIO] An I/O error occurs while reading from or writing to the file system.

45214 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path2*
 45215 argument.

45216 [ENAMETOOLONG]

45217 The length of the *path2* argument exceeds {PATH_MAX} or a pathname
 45218 component is longer than {NAME_MAX} or the length of the *path1* argument
 45219 is longer than {SYMLINK_MAX}.

45220 [ENOENT] A component of *path2* does not name an existing file or *path2* is an empty
 45221 string.

45222 [ENOSPC] The directory in which the entry for the new symbolic link is being placed
 45223 cannot be extended because no space is left on the file system containing the
 45224 directory, or the new symbolic link cannot be created because no space is left
 45225 on the file system which shall contain the link, or the file system is out of file-
 45226 allocation resources.

45227 [ENOTDIR] A component of the path prefix of *path2* is not a directory.

45228 [EROFS] The new symbolic link would reside on a read-only file system.

45229 The `symlink()` function may fail if:

45230 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 45231 resolution of the *path2* argument.

45232 [ENAMETOOLONG]

45233 As a result of encountering a symbolic link in resolution of the *path2*

45234 argument, the length of the substituted pathname string exceeded
45235 {PATH_MAX} bytes (including the terminating null byte), or the length of the
45236 string pointed to by *path1* exceeded {SYMLINK_MAX}.

45237 EXAMPLES

45238 None.

45239 APPLICATION USAGE

45240 Like a hard link, a symbolic link allows a file to have multiple logical names. The presence of a
45241 hard link guarantees the existence of a file, even after the original name has been removed. A
45242 symbolic link provides no such assurance; in fact, the file named by the *path1* argument need not
45243 exist when the link is created. A symbolic link can cross file system boundaries.

45244 Normal permission checks are made on each component of the symbolic link pathname during
45245 its resolution.

45246 RATIONALE

45247 Since IEEE Std 1003.1-2001 does not require any association of file times with symbolic links,
45248 there is no requirement that file times be updated by *symlink()*.

45249 FUTURE DIRECTIONS

45250 None.

45251 SEE ALSO

45252 *lchown()*, *link()*, *lstat()*, *open()*, *readlink()*, *unlink()*, the Base Definitions volume of
45253 IEEE Std 1003.1-2001, <unistd.h>

45254 CHANGE HISTORY

45255 First released in Issue 4, Version 2.

45256 Issue 5

45257 Moved from X/OPEN UNIX extension to BASE.

45258 Issue 6

45259 The following changes were made to align with the IEEE P1003.1a draft standard:

- 45260 • The DESCRIPTION text is updated.
- 45261 • The [ELOOP] optional error condition is added.

45262 **NAME**

45263 sync — schedule file system updates

45264 **SYNOPSIS**

45265 XSI #include <unistd.h>

45266 void sync(void);

45267

45268 **DESCRIPTION**45269 The *sync()* function shall cause all information in memory that updates file systems to be
45270 scheduled for writing out to all file systems.45271 The writing, although scheduled, is not necessarily complete upon return from *sync()*.45272 **RETURN VALUE**45273 The *sync()* function shall not return a value.45274 **ERRORS**

45275 No errors are defined.

45276 **EXAMPLES**

45277 None.

45278 **APPLICATION USAGE**

45279 None.

45280 **RATIONALE**

45281 None.

45282 **FUTURE DIRECTIONS**

45283 None.

45284 **SEE ALSO**45285 *fsync()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>45286 **CHANGE HISTORY**

45287 First released in Issue 4, Version 2.

45288 **Issue 5**

45289 Moved from X/OPEN UNIX extension to BASE.

45290 **NAME**

45291 sysconf — get configurable system variables

45292 **SYNOPSIS**

45293 #include <unistd.h>

45294 long sysconf(int name);

45295 **DESCRIPTION**

45296 The *sysconf()* function provides a method for the application to determine the current value of a
 45297 configurable system limit or option (*variable*). Support for some system variables is dependent
 45298 on implementation options (as indicated by the margin codes in the following table). Where an
 45299 implementation option is not supported, the variable need not be supported.

45300 The *name* argument represents the system variable to be queried. The following table lists the
 45301 minimal set of system variables from <limits.h> or <unistd.h> that can be returned by *sysconf()*,
 45302 and the symbolic constants defined in <unistd.h> that are the corresponding values used for
 45303 *name*. Support for some configuration variables is dependent on implementation options (see
 45304 shading and margin codes in the table below). Where an implementation option is not
 45305 supported, the variable need not be supported.

45306

45307

45308 AIO

45309

45310

45311

45312 XSI

45313

45314

45315

45316

45317

45318

45319

45320 XSI

45321

45322

45323 XSI

45324

45325

45326

45327 TSF

45328

45329

45330

45331 MSG

45332

Variable	Value of Name
{AIO_LISTIO_MAX}	_SC_AIO_LISTIO_MAX
{AIO_MAX}	_SC_AIO_MAX
{AIO_PRIO_DELTA_MAX}	_SC_AIO_PRIO_DELTA_MAX
{ARG_MAX}	_SC_ARG_MAX
{ATEXIT_MAX}	_SC_ATEXIT_MAX
{BC_BASE_MAX}	_SC_BC_BASE_MAX
{BC_DIM_MAX}	_SC_BC_DIM_MAX
{BC_SCALE_MAX}	_SC_BC_SCALE_MAX
{BC_STRING_MAX}	_SC_BC_STRING_MAX
{CHILD_MAX}	_SC_CHILD_MAX
Clock ticks/second	_SC_CLK_TCK
{COLL_WEIGHTS_MAX}	_SC_COLL_WEIGHTS_MAX
{DELAYTIMER_MAX}	_SC_DELAYTIMER_MAX
{EXPR_NEST_MAX}	_SC_EXPR_NEST_MAX
{HOST_NAME_MAX}	_SC_HOST_NAME_MAX
{IOV_MAX}	_SC_IOV_MAX
{LINE_MAX}	_SC_LINE_MAX
{LOGIN_NAME_MAX}	_SC_LOGIN_NAME_MAX
{NGROUPS_MAX}	_SC_NGROUPS_MAX
Maximum size of <i>getgrgid_r()</i> and <i>getgrnam_r()</i> data buffers	_SC_GETGR_R_SIZE_MAX
Maximum size of <i>getpwuid_r()</i> and <i>getpwnam_r()</i> data buffers	_SC_GETPW_R_SIZE_MAX
{MQ_OPEN_MAX}	_SC_MQ_OPEN_MAX
{MQ_PRIO_MAX}	_SC_MQ_PRIO_MAX

45333

45334

45335

45336 ADV

45337 BAR

45338 AIO

45339 CS

45340 CPT

45341

45342 FSC

45343

45344 MF

45345 ML

45346 MLR

45347 MPR

45348 MSG

45349 MON

45350

45351 PIO

45352 PS

45353 THR

45354 RTS

45355

45356

45357 SEM

45358 SHM

45359

45360 SPN

45361 SPI

45362 SS

45363 SIO

45364 TSA

45365 TSS

45366 TCT

45367 TPI

45368 TPP

45369 TPS

45370 TSH

45371 TSF

45372 TSP

45373 THR

45374 TMO

45375 TMR

45376 TRC

45377 TEF

45378 TRI

45379 TRL

45380 TYM

Variable	Value of Name
{OPEN_MAX}	_SC_OPEN_MAX
_POSIX_ADVISORY_INFO	_SC_ADVISORY_INFO
_POSIX_BARRIERS	_SC_BARRIERS
_POSIX_ASYNCHRONOUS_IO	_SC_ASYNCHRONOUS_IO
_POSIX_CLOCK_SELECTION	_SC_CLOCK_SELECTION
_POSIX_CPUTIME	_SC_CPUTIME
_POSIX_FILE_LOCKING	_SC_FILE_LOCKING
_POSIX_FSYNC	_SC_FSYNC
_POSIX_JOB_CONTROL	_SC_JOB_CONTROL
_POSIX_MAPPED_FILES	_SC_MAPPED_FILES
_POSIX_MEMLOCK	_SC_MEMLOCK
_POSIX_MEMLOCK_RANGE	_SC_MEMLOCK_RANGE
_POSIX_MEMORY_PROTECTION	_SC_MEMORY_PROTECTION
_POSIX_MESSAGE_PASSING	_SC_MESSAGE_PASSING
_POSIX_MONOTONIC_CLOCK	_SC_MONOTONIC_CLOCK
_POSIX_MULTI_PROCESS	_SC_MULTI_PROCESS
_POSIX_PRIORITIZED_IO	_SC_PRIORITIZED_IO
_POSIX_PRIORITY_SCHEDULING	_SC_PRIORITY_SCHEDULING
_POSIX_READER_WRITER_LOCKS	_SC_READER_WRITER_LOCKS
_POSIX_REALTIME_SIGNALS	_SC_REALTIME_SIGNALS
_POSIX_REGEX	_SC_REGEX
_POSIX_SAVED_IDS	_SC_SAVED_IDS
_POSIX_SEMAPHORES	_SC_SEMAPHORES
_POSIX_SHARED_MEMORY_OBJECTS	_SC_SHARED_MEMORY_OBJECTS
_POSIX_SHELL	_SC_SHELL
_POSIX_SPAWN	_SC_SPAWN
_POSIX_SPIN_LOCKS	_SC_SPIN_LOCKS
_POSIX_SPORADIC_SERVER	_SC_SPORADIC_SERVER
_POSIX_SYNCHRONIZED_IO	_SC_SYNCHRONIZED_IO
_POSIX_THREAD_ATTR_STACKADDR	_SC_THREAD_ATTR_STACKADDR
_POSIX_THREAD_ATTR_STACKSIZE	_SC_THREAD_ATTR_STACKSIZE
_POSIX_THREAD_CPUTIME	_SC_THREAD_CPUTIME
_POSIX_THREAD_PRIO_INHERIT	_SC_THREAD_PRIO_INHERIT
_POSIX_THREAD_PRIO_PROTECT	_SC_THREAD_PRIO_PROTECT
_POSIX_THREAD_PRIORITY_SCHEDULING	_SC_THREAD_PRIORITY_SCHEDULING
_POSIX_THREAD_PROCESS_SHARED	_SC_THREAD_PROCESS_SHARED
_POSIX_THREAD_SAFE_FUNCTIONS	_SC_THREAD_SAFE_FUNCTIONS
_POSIX_THREAD_SPORADIC_SERVER	_SC_THREAD_SPORADIC_SERVER
_POSIX_THREADS	_SC_THREADS
_POSIX_TIMEOUTS	_SC_TIMEOUTS
_POSIX_TIMERS	_SC_TIMERS
_POSIX_TRACE	_SC_TRACE
_POSIX_TRACE_EVENT_FILTER	_SC_TRACE_EVENT_FILTER
_POSIX_TRACE_INHERIT	_SC_TRACE_INHERIT
_POSIX_TRACE_LOG	_SC_TRACE_LOG
_POSIX_TYPED_MEMORY_OBJECTS	_SC_TYPED_MEMORY_OBJECTS

	Variable	Value of Name
45381	_POSIX_VERSION	_SC_VERSION
45382	_POSIX_V6_ILP32_OFF32	_SC_V6_ILP32_OFF32
45383	_POSIX_V6_ILP32_OFFBIG	_SC_V6_ILP32_OFFBIG
45384	_POSIX_V6_LP64_OFF64	_SC_V6_LP64_OFF64
45385	_POSIX_V6_LPBIG_OFFBIG	_SC_V6_LPBIG_OFFBIG
45386	_POSIX2_C_BIND	_SC_2_C_BIND
45387	_POSIX2_C_DEV	_SC_2_C_DEV
45388	_POSIX2_C_VERSION	_SC_2_C_VERSION
45389	_POSIX2_CHAR_TERM	_SC_2_CHAR_TERM
45390	_POSIX2_FORT_DEV	_SC_2_FORT_DEV
45391	_POSIX2_FORT_RUN	_SC_2_FORT_RUN
45392	_POSIX2_LOCALEDEF	_SC_2_LOCALEDEF
45393	_POSIX2_PBS	_SC_2_PBS
45394	_POSIX2_PBS_ACCOUNTING	_SC_2_PBS_ACCOUNTING
45395 BE	_POSIX2_PBS_LOCATE	_SC_2_PBS_LOCATE
45396	_POSIX2_PBS_MESSAGE	_SC_2_PBS_MESSAGE
45397	_POSIX2_PBS_TRACK	_SC_2_PBS_TRACK
45398	_POSIX2_SW_DEV	_SC_2_SW_DEV
45399	_POSIX2_UPE	_SC_2_UPE
45400	_POSIX2_VERSION	_SC_2_VERSION
45401	_REGEX_VERSION	_SC_REGEX_VERSION
45402	{PAGE_SIZE}	_SC_PAGE_SIZE
45403	{PAGESIZE}	_SC_PAGESIZE
45404	{PTHREAD_DESTRUCTOR_ITERATIONS}	_SC_THREAD_DESTRUCTOR_ITERATIONS
45405	{PTHREAD_KEYS_MAX}	_SC_THREAD_KEYS_MAX
45406 THR	{PTHREAD_STACK_MIN}	_SC_THREAD_STACK_MIN
45407	{PTHREAD_THREADS_MAX}	_SC_THREAD_THREADS_MAX
45408	{RE_DUP_MAX}	_SC_RE_DUP_MAX
45409	{RTSIG_MAX}	_SC_RTSIG_MAX
45410	{SEM_NSEMS_MAX}	_SC_SEM_NSEMS_MAX
45411 RTS	{SEM_VALUE_MAX}	_SC_SEM_VALUE_MAX
45412 SEM	{SIGQUEUE_MAX}	_SC_SIGQUEUE_MAX
45413	{STREAM_MAX}	_SC_STREAM_MAX
45414 RTS	{SYMLOOP_MAX}	_SC_SYMLOOP_MAX
45415	{TIMER_MAX}	_SC_TIMER_MAX
45416	{TTY_NAME_MAX}	_SC_TTY_NAME_MAX
45417 TMR	{TZNAME_MAX}	_SC_TZNAME_MAX
45418	_XBS5_ILP32_OFF32 (LEGACY)	_SC_XBS5_ILP32_OFF32 (LEGACY)
45419	_XBS5_ILP32_OFFBIG (LEGACY)	_SC_XBS5_ILP32_OFFBIG (LEGACY)
45420 XSI	_XBS5_LP64_OFF64 (LEGACY)	_SC_XBS5_LP64_OFF64 (LEGACY)
45421	_XBS5_LPBIG_OFFBIG (LEGACY)	_SC_XBS5_LPBIG_OFFBIG (LEGACY)
45422	_XOPEN_CRYPT	_SC_XOPEN_CRYPT
45423	_XOPEN_ENH_I18N	_SC_XOPEN_ENH_I18N
45424		
45425		

45426

45427

45428 XSI

45429

45430

45431

45432

45433

45434

Variable	Value of Name
_XOPEN_LEGACY	_SC_XOPEN_LEGACY
_XOPEN_REALTIME	_SC_XOPEN_REALTIME
_XOPEN_REALTIME_THREADS	_SC_XOPEN_REALTIME_THREADS
_XOPEN_SHM	_SC_XOPEN_SHM
_XOPEN_UNIX	_SC_XOPEN_UNIX
_XOPEN_VERSION	_SC_XOPEN_VERSION
_XOPEN_XCU_VERSION	_SC_XOPEN_XCU_VERSION

45435 **RETURN VALUE**

45436

45437

45438

If *name* is an invalid value, *sysconf()* shall return `-1` and set *errno* to indicate the error. If the variable corresponding to *name* has no limit, *sysconf()* shall return `-1` without changing the value of *errno*. Note that indefinite limits do not imply infinite limits; see `<limits.h>`.

45439

45440

45441

45442

Otherwise, *sysconf()* shall return the current variable value on the system. The value returned shall not be more restrictive than the corresponding value described to the application when it was compiled with the implementation's `<limits.h>` or `<unistd.h>`. The value shall not change during the lifetime of the calling process.

45443 **ERRORS**

45444

The *sysconf()* function shall fail if:

45445

[EINVAL] The value of the *name* argument is invalid.

45446 **EXAMPLES**

45447

None.

45448 **APPLICATION USAGE**

45449

45450

45451

As `-1` is a permissible return value in a successful situation, an application wishing to check for error situations should set *errno* to 0, then call *sysconf()*, and, if it returns `-1`, check to see if *errno* is non-zero.

45452

45453

45454

45455

45456

45457

45458

45459

If the value of *sysconf(_SC_2_VERSION)* is not equal to the value of the `_POSIX2_VERSION` symbolic constant, the utilities available via *system()* or *popen()* might not behave as described in the Shell and Utilities volume of IEEE Std 1003.1-2001. This would mean that the application is not running in an environment that conforms to the Shell and Utilities volume of IEEE Std 1003.1-2001. Some applications might be able to deal with this, others might not. However, the functions defined in this volume of IEEE Std 1003.1-2001 continue to operate as specified, even if *sysconf(_SC_2_VERSION)* reports that the utilities no longer perform as specified.

45460 **RATIONALE**

45461

45462

45463

This functionality was added in response to requirements of application developers and of system vendors who deal with many international system configurations. It is closely related to *pathconf()* and *fpathconf()*.

45464

45465

45466

45467

45468

Although a conforming application can run on all systems by never demanding more resources than the minimum values published in this volume of IEEE Std 1003.1-2001, it is useful for that application to be able to use the actual value for the quantity of a resource available on any given system. To do this, the application makes use of the value of a symbolic constant in `<limits.h>` or `<unistd.h>`.

45469

45470

45471

However, once compiled, the application must still be able to cope if the amount of resource available is increased. To that end, an application may need a means of determining the quantity of a resource, or the presence of an option, at execution time.

45472 Two examples are offered:

- 45473 1. Applications may wish to act differently on systems with or without job control.
 45474 Applications vendors who wish to distribute only a single binary package to all instances
 45475 of a computer architecture would be forced to assume job control is never available if it
 45476 were to rely solely on the `<unistd.h>` value published in this volume of
 45477 IEEE Std 1003.1-2001.
- 45478 2. International applications vendors occasionally require knowledge of the number of clock
 45479 ticks per second. Without these facilities, they would be required to either distribute their
 45480 applications partially in source form or to have 50 Hz and 60 Hz versions for the various
 45481 countries in which they operate.

45482 It is the knowledge that many applications are actually distributed widely in executable form
 45483 that leads to this facility. If limited to the most restrictive values in the headers, such
 45484 applications would have to be prepared to accept the most limited environments offered by the
 45485 smallest microcomputers. Although this is entirely portable, there was a consensus that they
 45486 should be able to take advantage of the facilities offered by large systems, without the
 45487 restrictions associated with source and object distributions.

45488 During the discussions of this feature, it was pointed out that it is almost always possible for an
 45489 application to discern what a value might be at runtime by suitably testing the various functions
 45490 themselves. And, in any event, it could always be written to adequately deal with error returns
 45491 from the various functions. In the end, it was felt that this imposed an unreasonable level of
 45492 complication and sophistication on the application writer.

45493 This runtime facility is not meant to provide ever-changing values that applications have to
 45494 check multiple times. The values are seen as changing no more frequently than once per system
 45495 initialization, such as by a system administrator or operator with an automatic configuration
 45496 program. This volume of IEEE Std 1003.1-2001 specifies that they shall not change within the
 45497 lifetime of the process.

45498 Some values apply to the system overall and others vary at the file system or directory level. The
 45499 latter are described in *pathconf()*.

45500 Note that all values returned must be expressible as integers. String values were considered, but
 45501 the additional flexibility of this approach was rejected due to its added complexity of
 45502 implementation and use.

45503 Some values, such as `{PATH_MAX}`, are sometimes so large that they must not be used to, say,
 45504 allocate arrays. The *sysconf()* function returns a negative value to show that this symbolic
 45505 constant is not even defined in this case.

45506 Similar to *pathconf()*, this permits the implementation not to have a limit. When one resource is
 45507 infinite, returning an error indicating that some other resource limit has been reached is
 45508 conforming behavior.

45509 FUTURE DIRECTIONS

45510 None.

45511 SEE ALSO

45512 *confstr()*, *pathconf()*, the Base Definitions volume of IEEE Std 1003.1-2001, `<limits.h>`,
 45513 `<unistd.h>`, the Shell and Utilities volume of IEEE Std 1003.1-2001, *getconf*

45514 CHANGE HISTORY

45515 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

45516 **Issue 5**

45517 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
 45518 Threads Extension.

45519 The `_XBS_` variables and name values are added to the table of system variables in the
 45520 DESCRIPTION. These are all marked EX.

45521 **Issue 6**

45522 The symbol `CLK_TCK` is obsolescent and removed. It is replaced with the phrase “clock ticks
 45523 per second”.

45524 The symbol `{PASS_MAX}` is removed.

45525 The following changes were made to align with the IEEE P1003.1a draft standard:

- 45526 • Table entries are added for the following variables: `_SC_REGEX`, `_SC_SHELL`,
 45527 `_SC_REGEX_VERSION`, `_SC_SYMLoop_MAX`.

45528 The following *sysconf()* variables and their associated names are added for alignment with
 45529 IEEE Std 1003.1d-1999:

```
45530     _POSIX_ADVISORY_INFO
45531     _POSIX_CPUTIME
45532     _POSIX_SPAWN
45533     _POSIX_SPORADIC_SERVER
45534     _POSIX_THREAD_CPUTIME
45535     _POSIX_THREAD_SPORADIC_SERVER
45536     _POSIX_TIMEOUTS
```

45537 The following changes are made to the DESCRIPTION for alignment with IEEE Std 1003.1j-2000:

- 45538 • A statement expressing the dependency of support for some system variables on
 45539 implementation options is added.
- 45540 • The following system variables are added:

```
45541     _POSIX_BARRIERS
45542     _POSIX_CLOCK_SELECTION
45543     _POSIX_MONOTONIC_CLOCK
45544     _POSIX_READER_WRITER_LOCKS
45545     _POSIX_SPIN_LOCKS
45546     _POSIX_TYPED_MEMORY_OBJECTS
```

45547 The following system variables are added for alignment with IEEE Std 1003.2d-1994:

```
45548     _POSIX2_PBS
45549     _POSIX2_PBS_ACCOUNTING
45550     _POSIX2_PBS_LOCATE
45551     _POSIX2_PBS_MESSAGE
45552     _POSIX2_PBS_TRACK
```

45553 The following *sysconf()* variables and their associated names are added for alignment with
 45554 IEEE Std 1003.1q-2000:

```
45555     _POSIX_TRACE
45556     _POSIX_TRACE_EVENT_FILTER
45557     _POSIX_TRACE_INHERIT
45558     _POSIX_TRACE_LOG
```


45559 The macros associated with the *c89* programming models are marked LEGACY, and new
45560 equivalent macros associated with *c99* are introduced.

45561 **NAME**

45562 syslog — log a message

45563 **SYNOPSIS**

45564 XSI #include <syslog.h>

45565 void syslog(int *priority*, const char **message*, ... /* *argument* */);

45566

45567 **DESCRIPTION**45568 Refer to *closelog()*.

45569 **NAME**

45570 system — issue a command

45571 **SYNOPSIS**

45572 #include <stdlib.h>

45573 int system(const char **command*);45574 **DESCRIPTION**

45575 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 45576 conflict between the requirements described here and the ISO C standard is unintentional. This
 45577 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

45578 If *command* is a null pointer, the *system()* function shall determine whether the host environment
 45579 has a command processor. If *command* is not a null pointer, the *system()* function shall pass the
 45580 string pointed to by *command* to that command processor to be executed in an implementation-
 45581 defined manner; this might then cause the program calling *system()* to behave in a non-
 45582 conforming manner or to terminate.

45583 CX The environment of the executed command shall be as if a child process were created using
 45584 *fork()*, and the child process invoked the *sh* utility using *execl()* as follows:

45585

```
execl(<shell path>, "sh", "-c", command, (char *)0);
```

45586 where <shell path> is an unspecified pathname for the *sh* utility.

45587 The *system()* function shall ignore the SIGINT and SIGQUIT signals, and shall block the
 45588 SIGCHLD signal, while waiting for the command to terminate. If this might cause the
 45589 application to miss a signal that would have killed it, then the application should examine the
 45590 return value from *system()* and take whatever action is appropriate to the application if the
 45591 command terminated due to receipt of a signal.

45592 The *system()* function shall not affect the termination status of any child of the calling processes
 45593 other than the process or processes it itself creates.

45594 The *system()* function shall not return until the child process has terminated.

45595 **RETURN VALUE**

45596 If *command* is a null pointer, *system()* shall return non-zero to indicate that a command processor
 45597 CX is available, or zero if none is available. The *system()* function shall always return non-zero when
 45598 *command* is NULL.

45599 CX If *command* is not a null pointer, *system()* shall return the termination status of the command
 45600 language interpreter in the format specified by *waitpid()*. The termination status shall be as
 45601 defined for the *sh* utility; otherwise, the termination status is unspecified. If some error prevents
 45602 the command language interpreter from executing after the child process is created, the return
 45603 value from *system()* shall be as if the command language interpreter had terminated using
 45604 *exit(127)* or *_exit(127)*. If a child process cannot be created, or if the termination status for the
 45605 command language interpreter cannot be obtained, *system()* shall return -1 and set *errno* to
 45606 indicate the error.

45607 **ERRORS**

45608 CX The *system()* function may set *errno* values as described by *fork()*.

45609 In addition, *system()* may fail if:

45610 CX [ECHILD] The status of the child process created by *system()* is no longer available.

45611 **EXAMPLES**

45612 None.

45613 **APPLICATION USAGE**

45614 If the return value of *system()* is not `-1`, its value can be decoded through the use of the macros
45615 described in `<sys/wait.h>`. For convenience, these macros are also provided in `<stdlib.h>`.

45616 Note that, while *system()* must ignore `SIGINT` and `SIGQUIT` and block `SIGCHLD` while waiting
45617 for the child to terminate, the handling of signals in the executed command is as specified by
45618 *fork()* and *exec*. For example, if `SIGINT` is being caught or is set to `SIG_DFL` when *system()* is
45619 called, then the child is started with `SIGINT` handling set to `SIG_DFL`.

45620 Ignoring `SIGINT` and `SIGQUIT` in the parent process prevents coordination problems (two
45621 processes reading from the same terminal, for example) when the executed command ignores or
45622 catches one of the signals. It is also usually the correct action when the user has given a
45623 command to the application to be executed synchronously (as in the `'!'` command in many
45624 interactive applications). In either case, the signal should be delivered only to the child process,
45625 not to the application itself. There is one situation where ignoring the signals might have less
45626 than the desired effect. This is when the application uses *system()* to perform some task invisible
45627 to the user. If the user typed the interrupt character ("`^C`", for example) while *system()* is being
45628 used in this way, one would expect the application to be killed, but only the executed command
45629 is killed. Applications that use *system()* in this way should carefully check the return status from
45630 *system()* to see if the executed command was successful, and should take appropriate action
45631 when the command fails.

45632 Blocking `SIGCHLD` while waiting for the child to terminate prevents the application from
45633 catching the signal and obtaining status from *system()*'s child process before *system()* can get the
45634 status itself.

45635 The context in which the utility is ultimately executed may differ from that in which *system()*
45636 was called. For example, file descriptors that have the `FD_CLOEXEC` flag set are closed, and the
45637 process ID and parent process ID are different. Also, if the executed utility changes its
45638 environment variables or its current working directory, that change is not reflected in the caller's
45639 context.

45640 There is no defined way for an application to find the specific path for the shell. However,
45641 *confstr()* can provide a value for *PATH* that is guaranteed to find the *sh* utility.

45642 **RATIONALE**

45643 The *system()* function should not be used by programs that have set user (or group) ID
45644 privileges. The *fork()* and *exec* family of functions (except *execlp()* and *execvp()*), should be used
45645 instead. This prevents any unforeseen manipulation of the environment of the user that could
45646 cause execution of commands not anticipated by the calling program.

45647 There are three levels of specification for the *system()* function. The ISO C standard gives the
45648 most basic. It requires that the function exists, and defines a way for an application to query
45649 whether a command language interpreter exists. It says nothing about the command language or
45650 the environment in which the command is interpreted.

45651 IEEE Std 1003.1-2001 places additional restrictions on *system()*. It requires that if there is a
45652 command language interpreter, the environment must be as specified by *fork()* and *exec*. This
45653 ensures, for example, that close-on-exec works, that file locks are not inherited, and that the
45654 process ID is different. It also specifies the return value from *system()* when the command line
45655 can be run, thus giving the application some information about the command's completion
45656 status.

Finally, IEEE Std 1003.1-2001 requires the command to be interpreted as in the shell command language defined in the Shell and Utilities volume of IEEE Std 1003.1-2001.

Note that, *system*(NULL) is required to return non-zero, indicating that there is a command language interpreter. At first glance, this would seem to conflict with the ISO C standard which allows *system*(NULL) to return zero. There is no conflict, however. A system must have a command language interpreter, and is non-conforming if none is present. It is therefore permissible for the *system*() function on such a system to implement the behavior specified by the ISO C standard as long as it is understood that the implementation does not conform to IEEE Std 1003.1-2001 if *system*(NULL) returns zero.

It was explicitly decided that when *command* is NULL, *system*() should not be required to check to make sure that the command language interpreter actually exists with the correct mode, that there are enough processes to execute it, and so on. The call *system*(NULL) could, theoretically, check for such problems as too many existing child processes, and return zero. However, it would be inappropriate to return zero due to such a (presumably) transient condition. If some condition exists that is not under the control of this application and that would cause any *system*() call to fail, that system has been rendered non-conforming.

Early drafts required, or allowed, *system*() to return with *errno* set to [EINTR] if it was interrupted with a signal. This error return was removed, and a requirement that *system*() not return until the child has terminated was added. This means that if a *waitpid*() call in *system*() exits with *errno* set to [EINTR], *system*() must reissue the *waitpid*(). This change was made for two reasons:

1. There is no way for an application to clean up if *system*() returns [EINTR], short of calling *wait*(), and that could have the undesirable effect of returning the status of children other than the one started by *system*().
2. While it might require a change in some historical implementations, those implementations already have to be changed because they use *wait*() instead of *waitpid*().

Note that if the application is catching SIGCHLD signals, it will receive such a signal before a successful *system*() call returns.

To conform to IEEE Std 1003.1-2001, *system*() must use *waitpid*(), or some similar function, instead of *wait*().

The following code sample illustrates how *system*() might be implemented on an implementation conforming to IEEE Std 1003.1-2001.

```
#include <signal.h>
int system(const char *cmd)
{
    int stat;
    pid_t pid;
    struct sigaction sa, savintr, savequit;
    sigset_t saveblock;
    if (cmd == NULL)
        return(1);
    sa.sa_handler = SIG_IGN;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigemptyset(&savintr.sa_mask);
    sigemptyset(&savequit.sa_mask);
    sigaction(SIGINT, &sa, &savintr);
    sigaction(SIGQUIT, &sa, &savequit);
```



```

45705     sigaddset(&sa.sa_mask, SIGCHLD);
45706     sigprocmask(SIG_BLOCK, &sa.sa_mask, &saveblock);
45707     if ((pid = fork()) == 0) {
45708         sigaction(SIGINT, &savintr, (struct sigaction *)0);
45709         sigaction(SIGQUIT, &savequit, (struct sigaction *)0);
45710         sigprocmask(SIG_SETMASK, &saveblock, (sigset_t *)0);
45711         execl("/bin/sh", "sh", "-c", cmd, (char *)0);
45712         _exit(127);
45713     }
45714     if (pid == -1) {
45715         stat = -1; /* errno comes from fork() */
45716     } else {
45717         while (waitpid(pid, &stat, 0) == -1) {
45718             if (errno != EINTR){
45719                 stat = -1;
45720                 break;
45721             }
45722         }
45723     }
45724     sigaction(SIGINT, &savintr, (struct sigaction *)0);
45725     sigaction(SIGQUIT, &savequit, (struct sigaction *)0);
45726     sigprocmask(SIG_SETMASK, &saveblock, (sigset_t *)0);
45727     return(stat);
45728 }

```

45729 Note that, while a particular implementation of *system()* (such as the one above) can assume a
 45730 particular path for the shell, such a path is not necessarily valid on another system. The above
 45731 example is not portable, and is not intended to be.

45732 One reviewer suggested that an implementation of *system()* might want to use an environment
 45733 variable such as *SHELL* to determine which command interpreter to use. The supposed
 45734 implementation would use the default command interpreter if the one specified by the
 45735 environment variable was not available. This would allow a user, when using an application
 45736 that prompts for command lines to be processed using *system()*, to specify a different command
 45737 interpreter. Such an implementation is discouraged. If the alternate command interpreter did not
 45738 follow the command line syntax specified in the Shell and Utilities volume of
 45739 IEEE Std 1003.1-2001, then changing *SHELL* would render *system()* non-conforming. This would
 45740 affect applications that expected the specified behavior from *system()*, and since the Shell and
 45741 Utilities volume of IEEE Std 1003.1-2001 does not mention that *SHELL* affects *system()*, the
 45742 application would not know that it needed to unset *SHELL*.

45743 FUTURE DIRECTIONS

45744 None.

45745 SEE ALSO

45746 *exec*, *pipe()*, *waitpid()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<limits.h>**,
 45747 **<signal.h>**, **<stdlib.h>**, **<sys/wait.h>**, the Shell and Utilities volume of IEEE Std 1003.1-2001, *sh*

45748 CHANGE HISTORY

45749 First released in Issue 1. Derived from Issue 1 of the SVID.

45750 Issue 6

45751 Extensions beyond the ISO C standard are marked.

45752 **NAME**

45753 tan, tanf, tanl — tangent function

45754 **SYNOPSIS**

45755 #include <math.h>

45756 double tan(double x);

45757 float tanf(float x);

45758 long double tanl(long double x);

45759 **DESCRIPTION**

45760 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 45761 conflict between the requirements described here and the ISO C standard is unintentional. This
 45762 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

45763 These functions shall compute the tangent of their argument *x*, measured in radians.

45764 An application wishing to check for error situations should set *errno* to zero and call
 45765 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 45766 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 45767 zero, an error has occurred.

45768 **RETURN VALUE**

45769 Upon successful completion, these functions shall return the tangent of *x*.

45770 If the correct value would cause underflow, and is not representable, a range error may occur,
 45771 MX and either 0.0 (if supported), or an implementation-defined value shall be returned.

45772 MX If *x* is NaN, a NaN shall be returned.

45773 If *x* is ± 0 , *x* shall be returned.

45774 If *x* is subnormal, a range error may occur and *x* should be returned.

45775 If *x* is $\pm\text{Inf}$, a domain error shall occur, and either a NaN (if supported), or an implementation-
 45776 defined value shall be returned.

45777 If the correct value would cause underflow, and is representable, a range error may occur and
 45778 the correct value shall be returned.

45779 XSI If the correct value would cause overflow, a range error shall occur and *tan()*, *tanf()*, and *tanl()*
 45780 shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

45781 **ERRORS**

45782 These functions shall fail if:

45783 MX **Domain Error** The value of *x* is $\pm\text{Inf}$.

45784 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 45785 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling
 45786 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 45787 shall be raised.

45788 XSI **Range Error** The result overflows

45789 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 45790 then *errno* shall be set to [ERANGE]. If the integer expression
 45791 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow
 45792 floating-point exception shall be raised.

45793 These functions may fail if:

45794 MX Range Error The result underflows, or the value of *x* is subnormal.

45795 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 45796 then *errno* shall be set to [ERANGE]. If the integer expression
 45797 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
 45798 floating-point exception shall be raised.

45799 EXAMPLES

45800 Taking the Tangent of a 45-Degree Angle

```
45801 #include <math.h>
45802 ...
45803 double radians = 45.0 * M_PI / 180;
45804 double result;
45805 ...
45806 result = tan (radians);
```

45807 APPLICATION USAGE

45808 There are no known floating-point representations such that for a normal argument, *tan*(*x*) is
 45809 either overflow or underflow.

45810 These functions may lose accuracy when their argument is near a multiple of $\pi/2$ or is far from
 45811 0.0.

45812 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
 45813 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

45814 RATIONALE

45815 None.

45816 FUTURE DIRECTIONS

45817 None.

45818 SEE ALSO

45819 *atan()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-2001,
 45820 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

45821 CHANGE HISTORY

45822 First released in Issue 1. Derived from Issue 1 of the SVID.

45823 Issue 5

45824 The last two paragraphs of the DESCRIPTION were included as APPLICATION USAGE notes
 45825 in previous issues.

45826 Issue 6

45827 The *tanf()* and *tanl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

45828 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
 45829 revised to align with the ISO/IEC 9899:1999 standard.

45830 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
 45831 marked.

45832 **NAME**

45833 tanh, tanhf, tanhl — hyperbolic tangent functions

45834 **SYNOPSIS**

45835 #include <math.h>

45836 double tanh(double x);

45837 float tanhf(float x);

45838 long double tanhl(long double x);

45839 **DESCRIPTION**

45840 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 45841 conflict between the requirements described here and the ISO C standard is unintentional. This
 45842 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

45843 These functions shall compute the hyperbolic tangent of their argument *x*.

45844 An application wishing to check for error situations should set *errno* to zero and call
 45845 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 45846 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 45847 zero, an error has occurred.

45848 **RETURN VALUE**45849 Upon successful completion, these functions shall return the hyperbolic tangent of *x*.45850 **MX** If *x* is NaN, a NaN shall be returned.45851 If *x* is ± 0 , *x* shall be returned.45852 If *x* is $\pm \text{Inf}$, ± 1 shall be returned.45853 If *x* is subnormal, a range error may occur and *x* should be returned.45854 **ERRORS**

45855 These functions may fail if:

45856 **MX** Range Error The value of *x* is subnormal.

45857 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 45858 then *errno* shall be set to [ERANGE]. If the integer expression
 45859 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
 45860 floating-point exception shall be raised.

45861 **EXAMPLES**

45862 None.

45863 **APPLICATION USAGE**

45864 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
 45865 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

45866 **RATIONALE**

45867 None.

45868 **FUTURE DIRECTIONS**

45869 None.

45870 **SEE ALSO**

45871 *atanh*(), *feclearexcept*(), *fetestexcept*(), *isnan*(), *tan*(), the Base Definitions volume of
 45872 IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
 45873 <math.h>

45874 **CHANGE HISTORY**

45875 First released in Issue 1. Derived from Issue 1 of the SVID.

45876 **Issue 5**

45877 The DESCRIPTION is updated to indicate how an application should check for an error. This
45878 text was previously published in the APPLICATION USAGE section.

45879 **Issue 6**

45880 The *tanhf()* and *tanhf()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

45881 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
45882 revised to align with the ISO/IEC 9899:1999 standard.

45883 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
45884 marked.

45885 **NAME**

45886 tanl — tangent function

45887 **SYNOPSIS**

45888 #include <math.h>

45889 long double tanl(long double x);

45890 **DESCRIPTION**45891 Refer to *tan()*.

45892 **NAME**

45893 tcdrain — wait for transmission of output

45894 **SYNOPSIS**

45895 #include <termios.h>

45896 int tcdrain(int *fil*des);45897 **DESCRIPTION**45898 The *tcdrain()* function shall block until all output written to the object referred to by *fil*des is transmitted. The *fil*des argument is an open file descriptor associated with a terminal.45900 Any attempts to use *tcdrain()* from a process which is a member of a background process group on a *fil*des associated with its controlling terminal, shall cause the process group to be sent a SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process shall be allowed to perform the operation, and no signal is sent.45904 **RETURN VALUE**45905 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to indicate the error.45907 **ERRORS**45908 The *tcdrain()* function shall fail if:45909 [EBADF] The *fil*des argument is not a valid file descriptor.45910 [EINTR] A signal interrupted *tcdrain()*.45911 [ENOTTY] The file associated with *fil*des is not a terminal.45912 The *tcdrain()* function may fail if:

45913 [EIO] The process group of the writing process is orphaned, and the writing process is not ignoring or blocking SIGTTOU.

45915 **EXAMPLES**

45916 None.

45917 **APPLICATION USAGE**

45918 None.

45919 **RATIONALE**

45920 None.

45921 **FUTURE DIRECTIONS**

45922 None.

45923 **SEE ALSO**45924 *tcflush()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface, <termios.h>, <unistd.h>45926 **CHANGE HISTORY**

45927 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

45928 **Issue 6**

45929 The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- 45931 • In the DESCRIPTION, the final paragraph is no longer conditional on
-
- 45932 _POSIX_JOB_CONTROL. This is a FIPS requirement.

45933

- The [EIO] error is added.

45934 **NAME**

45935 tcflow — suspend or restart the transmission or reception of data

45936 **SYNOPSIS**

45937 #include <termios.h>

45938 int tcflow(int *fildes*, int *action*);45939 **DESCRIPTION**

45940 The *tcflow()* function shall suspend or restart transmission or reception of data on the object
 45941 referred to by *fildes*, depending on the value of *action*. The *fildes* argument is an open file
 45942 descriptor associated with a terminal.

- 45943 • If *action* is TCOOFF, output shall be suspended.
- 45944 • If *action* is TCOON, suspended output shall be restarted.
- 45945 • If *action* is TCIOFF, the system shall transmit a STOP character, which is intended to cause
 45946 the terminal device to stop transmitting data to the system.
- 45947 • If *action* is TCION, the system shall transmit a START character, which is intended to cause
 45948 the terminal device to start transmitting data to the system.

45949 The default on the opening of a terminal file is that neither its input nor its output are
 45950 suspended.

45951 Attempts to use *tcflow()* from a process which is a member of a background process group on a
 45952 *fildes* associated with its controlling terminal, shall cause the process group to be sent a
 45953 SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process
 45954 shall be allowed to perform the operation, and no signal is sent.

45955 **RETURN VALUE**

45956 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
 45957 indicate the error.

45958 **ERRORS**45959 The *tcflow()* function shall fail if:

- 45960 [EBADF] The *fildes* argument is not a valid file descriptor.
- 45961 [EINVAL] The *action* argument is not a supported value.
- 45962 [ENOTTY] The file associated with *fildes* is not a terminal.

45963 The *tcflow()* function may fail if:

- 45964 [EIO] The process group of the writing process is orphaned, and the writing process
 45965 is not ignoring or blocking SIGTTOU.

45966 **EXAMPLES**

45967 None.

45968 **APPLICATION USAGE**

45969 None.

45970 **RATIONALE**

45971 None.

45972 **FUTURE DIRECTIONS**

45973 None.

45974 **SEE ALSO**

45975 *tcsendbreak()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11, General Terminal
45976 Interface, <termios.h>, <unistd.h>

45977 **CHANGE HISTORY**

45978 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

45979 **Issue 6**

45980 The following new requirements on POSIX implementations derive from alignment with the
45981 Single UNIX Specification:

- 45982 • The [EIO] error is added.

45983 **NAME**

45984 tcflush — flush non-transmitted output data, non-read input data, or both

45985 **SYNOPSIS**

45986 #include <termios.h>

45987 int tcflush(int *fildev*, int *queue_selector*);45988 **DESCRIPTION**

45989 Upon successful completion, *tcflush()* shall discard data written to the object referred to by *fildev*
 45990 (an open file descriptor associated with a terminal) but not transmitted, or data received but not
 45991 read, depending on the value of *queue_selector*:

- 45992 • If *queue_selector* is TCIFLUSH, it shall flush data received but not read.
- 45993 • If *queue_selector* is TCOFLUSH, it shall flush data written but not transmitted.
- 45994 • If *queue_selector* is TCIOFLUSH, it shall flush both data received but not read and data
 45995 written but not transmitted.

45996 Attempts to use *tcflush()* from a process which is a member of a background process group on a
 45997 *fildev* associated with its controlling terminal shall cause the process group to be sent a SIGTTOU
 45998 signal. If the calling process is blocking or ignoring SIGTTOU signals, the process shall be
 45999 allowed to perform the operation, and no signal is sent.

46000 **RETURN VALUE**

46001 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
 46002 indicate the error.

46003 **ERRORS**46004 The *tcflush()* function shall fail if:

- 46005 [EBADF] The *fildev* argument is not a valid file descriptor.
- 46006 [EINVAL] The *queue_selector* argument is not a supported value.
- 46007 [ENOTTY] The file associated with *fildev* is not a terminal.

46008 The *tcflush()* function may fail if:

- 46009 [EIO] The process group of the writing process is orphaned, and the writing process
 46010 is not ignoring or blocking SIGTTOU.

46011 **EXAMPLES**

46012 None.

46013 **APPLICATION USAGE**

46014 None.

46015 **RATIONALE**

46016 None.

46017 **FUTURE DIRECTIONS**

46018 None.

46019 **SEE ALSO**

46020 *tcdrain()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11, General Terminal
 46021 Interface, <termios.h>, <unistd.h>

46022 CHANGE HISTORY

46023 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

46024 Issue 6

46025 The Open Group Corrigendum U035/1 is applied. In the ERRORS and APPLICATION USAGE
46026 sections, references to *tcflow()* are replaced with *tcflush()*.

46027 The following new requirements on POSIX implementations derive from alignment with the
46028 Single UNIX Specification:

- 46029 • In the DESCRIPTION, the final paragraph is no longer conditional on
46030 `_POSIX_JOB_CONTROL`. This is a FIPS requirement.
- 46031 • The [EIO] error is added.

46032 **NAME**

46033 tcgetattr — get the parameters associated with the terminal

46034 **SYNOPSIS**

46035 #include <termios.h>

46036 int tcgetattr(int *fildes*, struct termios **termios_p*);46037 **DESCRIPTION**

46038 The *tcgetattr()* function shall get the parameters associated with the terminal referred to by *fildes*
46039 and store them in the **termios** structure referenced by *termios_p*. The *fildes* argument is an open
46040 file descriptor associated with a terminal.

46041 The *termios_p* argument is a pointer to a **termios** structure.46042 The *tcgetattr()* operation is allowed from any process.

46043 If the terminal device supports different input and output baud rates, the baud rates stored in
46044 the **termios** structure returned by *tcgetattr()* shall reflect the actual baud rates, even if they are
46045 equal. If differing baud rates are not supported, the rate returned as the output baud rate shall be
46046 the actual baud rate. If the terminal device does not support split baud rates, the input baud rate
46047 stored in the **termios** structure shall be the output rate (as one of the symbolic values).

46048 **RETURN VALUE**

46049 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
46050 indicate the error.

46051 **ERRORS**46052 The *tcgetattr()* function shall fail if:46053 [EBADF] The *fildes* argument is not a valid file descriptor.46054 [ENOTTY] The file associated with *fildes* is not a terminal.46055 **EXAMPLES**

46056 None.

46057 **APPLICATION USAGE**

46058 None.

46059 **RATIONALE**

46060 Care must be taken when changing the terminal attributes. Applications should always do a
46061 *tcgetattr()*, save the **termios** structure values returned, and then do a *tcsetattr()*, changing only
46062 the necessary fields. The application should use the values saved from the *tcgetattr()* to reset the
46063 terminal state whenever it is done with the terminal. This is necessary because terminal
46064 attributes apply to the underlying port and not to each individual open instance; that is, all
46065 processes that have used the terminal see the latest attribute changes.

46066 A program that uses these functions should be written to catch all signals and take other
46067 appropriate actions to ensure that when the program terminates, whether planned or not, the
46068 terminal device's state is restored to its original state.

46069 Existing practice dealing with error returns when only part of a request can be honored is based
46070 on calls to the *ioctl()* function. In historical BSD and System V implementations, the
46071 corresponding *ioctl()* returns zero if the requested actions were semantically correct, even if
46072 some of the requested changes could not be made. Many existing applications assume this
46073 behavior and would no longer work correctly if the return value were changed from zero to -1
46074 in this case.

46075 Note that either specification has a problem. When zero is returned, it implies everything
46076 succeeded even if some of the changes were not made. When -1 is returned, it implies
46077 everything failed even though some of the changes were made.

46078 Applications that need all of the requested changes made to work properly should follow
46079 *tcsetattr()* with a call to *tcgetattr()* and compare the appropriate field values.

46080 **FUTURE DIRECTIONS**

46081 None.

46082 **SEE ALSO**

46083 *tcsetattr()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11, General Terminal
46084 Interface, <**termios.h**>

46085 **CHANGE HISTORY**

46086 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

46087 **Issue 6**

46088 In the DESCRIPTION, the rate returned as the input baud rate shall be the output rate.
46089 Previously, the number zero was also allowed but was obsolescent.

46090 **NAME**

46091 tcgetpgrp — get the foreground process group ID

46092 **SYNOPSIS**

46093 #include <unistd.h>

46094 pid_t tcgetpgrp(int *fildev*);46095 **DESCRIPTION**46096 The *tcgetpgrp()* function shall return the value of the process group ID of the foreground process
46097 group associated with the terminal.46098 If there is no foreground process group, *tcgetpgrp()* shall return a value greater than 1 that does
46099 not match the process group ID of any existing process group.46100 The *tcgetpgrp()* function is allowed from a process that is a member of a background process
46101 group; however, the information may be subsequently changed by a process that is a member of
46102 a foreground process group.46103 **RETURN VALUE**46104 Upon successful completion, *tcgetpgrp()* shall return the value of the process group ID of the
46105 foreground process associated with the terminal. Otherwise, -1 shall be returned and *errno* set to
46106 indicate the error.46107 **ERRORS**46108 The *tcgetpgrp()* function shall fail if:46109 [EBADF] The *fildev* argument is not a valid file descriptor.46110 [ENOTTY] The calling process does not have a controlling terminal, or the file is not the
46111 controlling terminal.46112 **EXAMPLES**

46113 None.

46114 **APPLICATION USAGE**

46115 None.

46116 **RATIONALE**

46117 None.

46118 **FUTURE DIRECTIONS**

46119 None.

46120 **SEE ALSO**46121 *setsid()*, *setpgid()*, *tcsetpgrp()*, the Base Definitions volume of IEEE Std 1003.1-2001,
46122 <sys/types.h>, <unistd.h>46123 **CHANGE HISTORY**

46124 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

46125 **Issue 6**

46126 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

46127 The following new requirements on POSIX implementations derive from alignment with the
46128 Single UNIX Specification:

- 46129
- The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
46130 required for conforming implementations of previous POSIX specifications, it was not
46131 required for UNIX applications.

46132

46133

- In the DESCRIPTION, text previously conditional on support for _POSIX_JOB_CONTROL is now mandatory. This is a FIPS requirement.

46134 NAME

46135 `tcgetsid` — get the process group ID for the session leader for the controlling terminal

46136 SYNOPSIS

46137 XSI `#include <termios.h>`

46138 `pid_t tcgetsid(int fildes);`

46139

46140 DESCRIPTION

46141 The `tcgetsid()` function shall obtain the process group ID of the session for which the terminal
46142 specified by *fildes* is the controlling terminal.

46143 RETURN VALUE

46144 Upon successful completion, `tcgetsid()` shall return the process group ID associated with the
46145 terminal. Otherwise, a value of `(pid_t)-1` shall be returned and *errno* set to indicate the error.

46146 ERRORS

46147 The `tcgetsid()` function shall fail if:

46148 [EBADF] The *fildes* argument is not a valid file descriptor.

46149 [ENOTTY] The calling process does not have a controlling terminal, or the file is not the
46150 controlling terminal.

46151 EXAMPLES

46152 None.

46153 APPLICATION USAGE

46154 None.

46155 RATIONALE

46156 None.

46157 FUTURE DIRECTIONS

46158 None.

46159 SEE ALSO

46160 The Base Definitions volume of IEEE Std 1003.1-2001, `<termios.h>`

46161 CHANGE HISTORY

46162 First released in Issue 4, Version 2.

46163 Issue 5

46164 Moved from X/OPEN UNIX extension to BASE.

46165 The [EACCES] error has been removed from the list of mandatory errors, and the description of
46166 [ENOTTY] has been reworded.

46167 **NAME**

46168 tcsendbreak — send a break for a specific duration

46169 **SYNOPSIS**

46170 #include <termios.h>

46171 int tcsendbreak(int *fildev*, int *duration*);46172 **DESCRIPTION**

46173 If the terminal is using asynchronous serial data transmission, *tcsendbreak()* shall cause
46174 transmission of a continuous stream of zero-valued bits for a specific duration. If *duration* is 0, it
46175 shall cause transmission of zero-valued bits for at least 0.25 seconds, and not more than 0.5
46176 seconds. If *duration* is not 0, it shall send zero-valued bits for an implementation-defined period
46177 of time.

46178 The *fildev* argument is an open file descriptor associated with a terminal.

46179 If the terminal is not using asynchronous serial data transmission, it is implementation-defined
46180 whether *tcsendbreak()* sends data to generate a break condition or returns without taking any
46181 action.

46182 Attempts to use *tcsendbreak()* from a process which is a member of a background process group
46183 on a *fildev* associated with its controlling terminal shall cause the process group to be sent a
46184 SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process
46185 shall be allowed to perform the operation, and no signal is sent.

46186 **RETURN VALUE**

46187 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
46188 indicate the error.

46189 **ERRORS**

46190 The *tcsendbreak()* function shall fail if:

46191 [EBADF] The *fildev* argument is not a valid file descriptor.

46192 [ENOTTY] The file associated with *fildev* is not a terminal.

46193 The *tcsendbreak()* function may fail if:

46194 [EIO] The process group of the writing process is orphaned, and the writing process
46195 is not ignoring or blocking SIGTTOU.

46196 **EXAMPLES**

46197 None.

46198 **APPLICATION USAGE**

46199 None.

46200 **RATIONALE**

46201 None.

46202 **FUTURE DIRECTIONS**

46203 None.

46204 **SEE ALSO**

46205 The Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface,
46206 <termios.h>, <unistd.h>

46207 CHANGE HISTORY

46208 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

46209 Issue 6

46210 The following new requirements on POSIX implementations derive from alignment with the
46211 Single UNIX Specification:

- 46212 • In the DESCRIPTION, text previously conditional on _POSIX_JOB_CONTROL is now
- 46213 mandated. This is a FIPS requirement.
- 46214 • The [EIO] error is added.

46215 NAME

46216 tcsetattr — set the parameters associated with the terminal

46217 SYNOPSIS

46218 #include <termios.h>

```
46219 int tcsetattr(int fildes, int optional_actions,
46220               const struct termios *termios_p);
```

46221 DESCRIPTION

46222 The *tcsetattr()* function shall set the parameters associated with the terminal referred to by the
 46223 open file descriptor *fildes* (an open file descriptor associated with a terminal) from the **termios**
 46224 structure referenced by *termios_p* as follows:

- 46225 • If *optional_actions* is TCSANOW, the change shall occur immediately.
- 46226 • If *optional_actions* is TCSADRAIN, the change shall occur after all output written to *fildes* is
 46227 transmitted. This function should be used when changing parameters that affect output.
- 46228 • If *optional_actions* is TCSAFLUSH, the change shall occur after all output written to *fildes* is
 46229 transmitted, and all input so far received but not read shall be discarded before the change is
 46230 made.

46231 If the output baud rate stored in the **termios** structure pointed to by *termios_p* is the zero baud
 46232 rate, B0, the modem control lines shall no longer be asserted. Normally, this shall disconnect the
 46233 line.

46234 If the input baud rate stored in the **termios** structure pointed to by *termios_p* is 0, the input baud
 46235 rate given to the hardware is the same as the output baud rate stored in the **termios** structure.

46236 The *tcsetattr()* function shall return successfully if it was able to perform any of the requested
 46237 actions, even if some of the requested actions could not be performed. It shall set all the
 46238 attributes that the implementation supports as requested and leave all the attributes not
 46239 supported by the implementation unchanged. If no part of the request can be honored, it shall
 46240 return `-1` and set *errno* to `[EINVAL]`. If the input and output baud rates differ and are a
 46241 combination that is not supported, neither baud rate shall be changed. A subsequent call to
 46242 *tcgetattr()* shall return the actual state of the terminal device (reflecting both the changes made
 46243 and not made in the previous *tcsetattr()* call). The *tcsetattr()* function shall not change the values
 46244 found in the **termios** structure under any circumstances.

46245 The effect of *tcsetattr()* is undefined if the value of the **termios** structure pointed to by *termios_p*
 46246 was not derived from the result of a call to *tcgetattr()* on *fildes*; an application should modify
 46247 only fields and flags defined by this volume of IEEE Std 1003.1-2001 between the call to
 46248 *tcgetattr()* and *tcsetattr()*, leaving all other fields and flags unmodified.

46249 No actions defined by this volume of IEEE Std 1003.1-2001, other than a call to *tcsetattr()* or a
 46250 close of the last file descriptor in the system associated with this terminal device, shall cause any
 46251 of the terminal attributes defined by this volume of IEEE Std 1003.1-2001 to change.

46252 If *tcsetattr()* is called from a process which is a member of a background process group on a
 46253 *fildes* associated with its controlling terminal:

- 46254 • If the calling process is blocking or ignoring SIGTTOU signals, the operation completes
 46255 normally and no signal is sent.
- 46256 • Otherwise, a SIGTTOU signal shall be sent to the process group.

46257 RETURN VALUE

46258 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
 46259 indicate the error.

46260 ERRORS

46261 The *tcsetattr()* function shall fail if:

46262 [EBADF] The *fildev* argument is not a valid file descriptor.

46263 [EINTR] A signal interrupted *tcsetattr()*.

46264 [EINVAL] The *optional_actions* argument is not a supported value, or an attempt was
 46265 made to change an attribute represented in the **termios** structure to an
 46266 unsupported value.

46267 [ENOTTY] The file associated with *fildev* is not a terminal.

46268 The *tcsetattr()* function may fail if:

46269 [EIO] The process group of the writing process is orphaned, and the writing process
 46270 is not ignoring or blocking SIGTTOU.

46271 EXAMPLES

46272 None.

46273 APPLICATION USAGE

46274 If trying to change baud rates, applications should call *tcsetattr()* then call *tcgetattr()* in order to
 46275 determine what baud rates were actually selected.

46276 RATIONALE

46277 The *tcsetattr()* function can be interrupted in the following situations:

- 46278 • It is interrupted while waiting for output to drain.
- 46279 • It is called from a process in a background process group and SIGTTOU is caught.

46280 See also the RATIONALE section in *tcgetattr()*.

46281 FUTURE DIRECTIONS

46282 Using an input baud rate of 0 to set the input rate equal to the output rate may not necessarily be
 46283 supported in a future version of this volume of IEEE Std 1003.1-2001.

46284 SEE ALSO

46285 *cfgetispeed()*, *tcgetattr()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11,
 46286 General Terminal Interface, <termios.h>, <unistd.h>

46287 CHANGE HISTORY

46288 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

46289 Issue 6

46290 The following new requirements on POSIX implementations derive from alignment with the
 46291 Single UNIX Specification:

- 46292 • In the DESCRIPTION, text previously conditional on _POSIX_JOB_CONTROL is now
 46293 mandated. This is a FIPS requirement.
- 46294 • The [EIO] error is added.

46295 In the DESCRIPTION, the text describing use of *tcsetattr()* from a process which is a member of
46296 a background process group is clarified.

46297 **NAME**

46298 tcsetpgrp — set the foreground process group ID

46299 **SYNOPSIS**

46300 #include <unistd.h>

46301 int tcsetpgrp(int *fildev*, pid_t *pgid_id*);46302 **DESCRIPTION**

46303 If the process has a controlling terminal, *tcsetpgrp()* shall set the foreground process group ID
 46304 associated with the terminal to *pgid_id*. The application shall ensure that the file associated with
 46305 *fildev* is the controlling terminal of the calling process and the controlling terminal is currently
 46306 associated with the session of the calling process. The application shall ensure that the value of
 46307 *pgid_id* matches a process group ID of a process in the same session as the calling process.

46308 Attempts to use *tcsetpgrp()* from a process which is a member of a background process group on
 46309 a *fildev* associated with its controlling terminal shall cause the process group to be sent a
 46310 SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process
 46311 shall be allowed to perform the operation, and no signal is sent.

46312 **RETURN VALUE**

46313 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
 46314 indicate the error.

46315 **ERRORS**46316 The *tcsetpgrp()* function shall fail if:46317 [EBADF] The *fildev* argument is not a valid file descriptor.46318 [EINVAL] This implementation does not support the value in the *pgid_id* argument.

46319 [ENOTTY] The calling process does not have a controlling terminal, or the file is not the
 46320 controlling terminal, or the controlling terminal is no longer associated with
 46321 the session of the calling process.

46322 [EPERM] The value of *pgid_id* is a value supported by the implementation, but does not
 46323 match the process group ID of a process in the same session as the calling
 46324 process.

46325 **EXAMPLES**

46326 None.

46327 **APPLICATION USAGE**

46328 None.

46329 **RATIONALE**

46330 None.

46331 **FUTURE DIRECTIONS**

46332 None.

46333 **SEE ALSO**46334 *tcgetpgrp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>46335 **CHANGE HISTORY**

46336 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

46337 **Issue 6**

46338 In the SYNOPSIS, the inclusion of `<sys/types.h>` is no longer required.

46339 The following new requirements on POSIX implementations derive from alignment with the
46340 Single UNIX Specification:

46341 • The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was
46342 required for conforming implementations of previous POSIX specifications, it was not
46343 required for UNIX applications.

46344 • In the DESCRIPTION and ERRORS sections, text previously conditional on
46345 `_POSIX_JOB_CONTROL` is now mandated. This is a FIPS requirement.

46346 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

46347 The Open Group Corrigendum U047/4 is applied.

46348 NAME

46349 tdelete, tfind, tsearch, twalk — manage a binary search tree

46350 SYNOPSIS

```

46351 xsi      #include <search.h>

46352 void *tdelete(const void *restrict key, void **restrict rootp,
46353             int(*compar)(const void *, const void *));
46354 void *tfind(const void *key, void *const *rootp,
46355            int(*compar)(const void *, const void *));
46356 void *tsearch(const void *key, void **rootp,
46357              int (*compar)(const void *, const void *));
46358 void twalk(const void *root,
46359           void (*action)(const void *, VISIT, int));
46360

```

46361 DESCRIPTION

46362 The *tdelete()*, *tfind()*, *tsearch()*, and *twalk()* functions manipulate binary search trees.
 46363 Comparisons are made with a user-supplied routine, the address of which is passed as the
 46364 *compar* argument. This routine is called with two arguments, which are the pointers to the
 46365 elements being compared. The application shall ensure that the user-supplied routine returns an
 46366 integer less than, equal to, or greater than 0, according to whether the first argument is to be
 46367 considered less than, equal to, or greater than the second argument. The comparison function
 46368 need not compare every byte, so arbitrary data may be contained in the elements in addition to
 46369 the values being compared.

46370 The *tsearch()* function shall build and access the tree. The *key* argument is a pointer to an element
 46371 to be accessed or stored. If there is a node in the tree whose element is equal to the value pointed
 46372 to by *key*, a pointer to this found node shall be returned. Otherwise, the value pointed to by *key*
 46373 shall be inserted (that is, a new node is created and the value of *key* is copied to this node), and a
 46374 pointer to this node returned. Only pointers are copied, so the application shall ensure that the
 46375 calling routine stores the data. The *rootp* argument points to a variable that points to the root
 46376 node of the tree. A null pointer value for the variable pointed to by *rootp* denotes an empty tree;
 46377 in this case, the variable shall be set to point to the node which shall be at the root of the new
 46378 tree.

46379 Like *tsearch()*, *tfind()* shall search for a node in the tree, returning a pointer to it if found.
 46380 However, if it is not found, *tfind()* shall return a null pointer. The arguments for *tfind()* are the
 46381 same as for *tsearch()*.

46382 The *tdelete()* function shall delete a node from a binary search tree. The arguments are the same
 46383 as for *tsearch()*. The variable pointed to by *rootp* shall be changed if the deleted node was the
 46384 root of the tree. The *tdelete()* function shall return a pointer to the parent of the deleted node, or a
 46385 null pointer if the node is not found.

46386 The *twalk()* function shall traverse a binary search tree. The *root* argument is a pointer to the root
 46387 node of the tree to be traversed. (Any node in a tree may be used as the root for a walk below
 46388 that node.) The argument *action* is the name of a routine to be invoked at each node. This routine
 46389 is, in turn, called with three arguments. The first argument shall be the address of the node being
 46390 visited. The structure pointed to by this argument is unspecified and shall not be modified by
 46391 the application, but it shall be possible to cast a pointer-to-node into a pointer-to-pointer-to-
 46392 element to access the element stored in the node. The second argument shall be a value from an
 46393 enumeration data type:

```

46394 typedef enum { preorder, postorder, endorder, leaf } VISIT;

```


46395 (defined in `<search.h>`), depending on whether this is the first, second, or third time that the
 46396 node is visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a
 46397 leaf. The third argument shall be the level of the node in the tree, with the root being level 0.

46398 If the calling function alters the pointer to the root, the result is undefined.

46399 RETURN VALUE

46400 If the node is found, both `tsearch()` and `tfind()` shall return a pointer to it. If not, `tfind()` shall
 46401 return a null pointer, and `tsearch()` shall return a pointer to the inserted item.

46402 A null pointer shall be returned by `tsearch()` if there is not enough space available to create a new
 46403 node.

46404 A null pointer shall be returned by `tdelete()`, `tfind()`, and `tsearch()` if `rootp` is a null pointer on
 46405 entry.

46406 The `tdelete()` function shall return a pointer to the parent of the deleted node, or a null pointer if
 46407 the node is not found.

46408 The `twalk()` function shall not return a value.

46409 ERRORS

46410 No errors are defined.

46411 EXAMPLES

46412 The following code reads in strings and stores structures containing a pointer to each string and
 46413 a count of its length. It then walks the tree, printing out the stored strings and their lengths in
 46414 alphabetical order.

```
46415 #include <search.h>
46416 #include <string.h>
46417 #include <stdio.h>

46418 #define STRSZ    10000
46419 #define NODSZ    500

46420 struct node {      /* Pointers to these are stored in the tree. */
46421     char    *string;
46422     int     length;
46423 };

46424 char    string_space[STRSZ]; /* Space to store strings. */
46425 struct node nodes[NODSZ];    /* Nodes to store. */
46426 void    *root = NULL;        /* This points to the root. */

46427 int main(int argc, char *argv[])
46428 {
46429     char    *strptr = string_space;
46430     struct node *nodeptr = nodes;
46431     void    print_node(const void *, VISIT, int);
46432     int     i = 0, node_compare(const void *, const void *);

46433     while (gets(strptr) != NULL && i++ < NODSZ) {
46434         /* Set node. */
46435         nodeptr->string = strptr;
46436         nodeptr->length = strlen(strptr);
46437         /* Put node into the tree. */
46438         (void) tsearch((void *)nodeptr, (void **)&root,
46439             node_compare);
```



```

46440         /* Adjust pointers, so we do not overwrite tree. */
46441         strptr += nodeptr->length + 1;
46442         nodeptr++;
46443     }
46444     twalk(root, print_node);
46445     return 0;
46446 }

46447 /*
46448  * This routine compares two nodes, based on an
46449  * alphabetical ordering of the string field.
46450  */
46451 int
46452 node_compare(const void *node1, const void *node2)
46453 {
46454     return strcmp(((const struct node *) node1)->string,
46455                  ((const struct node *) node2)->string);
46456 }

46457 /*
46458  * This routine prints out a node, the second time
46459  * twalk encounters it or if it is a leaf.
46460  */
46461 void
46462 print_node(const void *ptr, VISIT order, int level)
46463 {
46464     const struct node *p = *(const struct node **) ptr;

46465     if (order == postorder || order == leaf) {
46466         (void) printf("string = %s, length = %d\n",
46467                     p->string, p->length);
46468     }
46469 }

```

46470 APPLICATION USAGE

46471 The *root* argument to *twalk()* is one level of indirection less than the *rootp* arguments to *tdelete()*
 46472 and *tsearch()*.

46473 There are two nomenclatures used to refer to the order in which tree nodes are visited. The
 46474 *tsearch()* function uses **preorder**, **postorder**, and **endorder** to refer respectively to visiting a node
 46475 before any of its children, after its left child and before its right, and after both its children. The
 46476 alternative nomenclature uses **preorder**, **inorder**, and **postorder** to refer to the same visits, which
 46477 could result in some confusion over the meaning of **postorder**.

46478 RATIONALE

46479 None.

46480 FUTURE DIRECTIONS

46481 None.

46482 SEE ALSO

46483 *hcreate()*, *tsearch()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**search.h**>

46484 CHANGE HISTORY

46485 First released in Issue 1. Derived from Issue 1 of the SVID.

46486 Issue 5

46487 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in
46488 previous issues.

46489 Issue 6

46490 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

46491 The **restrict** keyword is added to the *tdelete()* prototype for alignment with the
46492 ISO/IEC 9899:1999 standard.

46493 **NAME**

46494 telldir — current location of a named directory stream

46495 **SYNOPSIS**46496 XSI `#include <dirent.h>`46497 `long telldir(DIR *dirp);`

46498

46499 **DESCRIPTION**46500 The *telldir()* function shall obtain the current location associated with the directory stream
46501 specified by *dirp*.46502 If the most recent operation on the directory stream was a *seekdir()*, the directory position
46503 returned from the *telldir()* shall be the same as that supplied as a *loc* argument for *seekdir()*.46504 **RETURN VALUE**46505 Upon successful completion, *telldir()* shall return the current location of the specified directory
46506 stream.46507 **ERRORS**

46508 No errors are defined.

46509 **EXAMPLES**

46510 None.

46511 **APPLICATION USAGE**

46512 None.

46513 **RATIONALE**

46514 None.

46515 **FUTURE DIRECTIONS**

46516 None.

46517 **SEE ALSO**46518 *opendir()*, *readdir()*, *seekdir()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<dirent.h>**46519 **CHANGE HISTORY**

46520 First released in Issue 2.

46521 NAME

46522 tempnam — create a name for a temporary file

46523 SYNOPSIS

46524 XSI `#include <stdio.h>`46525 `char *tempnam(const char *dir, const char *pfx);`

46526

46527 DESCRIPTION

46528 The *tempnam()* function shall generate a pathname that may be used for a temporary file.

46529 The *tempnam()* function allows the user to control the choice of a directory. The *dir* argument
 46530 points to the name of the directory in which the file is to be created. If *dir* is a null pointer or
 46531 points to a string which is not a name for an appropriate directory, the path prefix defined as
 46532 P_tmpdir in the <stdio.h> header shall be used. If that directory is not accessible, an
 46533 implementation-defined directory may be used.

46534 Many applications prefer their temporary files to have certain initial letter sequences in their
 46535 names. The *pfx* argument should be used for this. This argument may be a null pointer or point
 46536 to a string of up to five bytes to be used as the beginning of the filename.

46537 Some implementations of *tempnam()* may use *tmpnam()* internally. On such implementations, if
 46538 called more than {TMP_MAX} times in a single process, the behavior is implementation-defined.

46539 RETURN VALUE

46540 Upon successful completion, *tempnam()* shall allocate space for a string, put the generated
 46541 pathname in that space, and return a pointer to it. The pointer shall be suitable for use in a
 46542 subsequent call to *free()*. Otherwise, it shall return a null pointer and set *errno* to indicate the
 46543 error.

46544 ERRORS

46545 The *tempnam()* function shall fail if:

46546 [ENOMEM] Insufficient storage space is available.

46547 EXAMPLES

46548 Generating a Pathname

46549 The following example generates a pathname for a temporary file in directory */tmp*, with the
 46550 prefix *file*. After the filename has been created, the call to *free()* deallocates the space used to
 46551 store the filename.

```
46552 #include <stdio.h>
46553 #include <stdlib.h>
46554 ...
46555 char *directory = "/tmp";
46556 char *fileprefix = "file";
46557 char *file;

46558 file = tempnam(directory, fileprefix);
46559 free(file);
```

46560 APPLICATION USAGE

46561 This function only creates pathnames. It is the application's responsibility to create and remove
 46562 the files. Between the time a pathname is created and the file is opened, it is possible for some
 46563 other process to create a file with the same name. Applications may find *tmpfile()* more useful.

46564 **RATIONALE**

46565 None.

46566 **FUTURE DIRECTIONS**

46567 None.

46568 **SEE ALSO**

46569 *fopen()*, *free()*, *open()*, *tmpfile()*, *tmpnam()*, *unlink()*, the Base Definitions volume of
46570 IEEE Std 1003.1-2001, <**stdio.h**>

46571 **CHANGE HISTORY**

46572 First released in Issue 1. Derived from Issue 1 of the SVID.

46573 **Issue 5**

46574 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in
46575 previous issues.

46576 NAME

46577 tfind — search binary search tree

46578 SYNOPSIS

46579 XSI #include <search.h>

46580 void *tfind(const void *key, void *const *rootp,
46581 int (*compar)(const void *, const void *));

46582

46583 DESCRIPTION

46584 Refer to *tdelete()*.

46585 **NAME**

46586 tgamma, tgammaf, tgammal — compute gamma() function

46587 **SYNOPSIS**

46588 #include <math.h>

46589 double tgamma(double x);

46590 float tgammaf(float x);

46591 long double tgammal(long double x);

46592 **DESCRIPTION**

46593 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 46594 conflict between the requirements described here and the ISO C standard is unintentional. This
 46595 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

46596 These functions shall compute the *gamma()* function of *x*.

46597 An application wishing to check for error situations should set *errno* to zero and call
 46598 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 46599 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 46600 zero, an error has occurred.

46601 **RETURN VALUE**46602 Upon successful completion, these functions shall return *Gamma(x)*.

46603 If *x* is a negative integer, a domain error shall occur, and either a NaN (if supported), or an
 46604 implementation-defined value shall be returned.

46605 If the correct value would cause overflow, a range error shall occur and *tgamma()*, *tgammaf()*,
 46606 and *tgammal()* shall return the value of the macro HUGE_VAL, HUGE_VALF, or HUGE_VALL,
 46607 respectively.

46608 **MX** If *x* is NaN, a NaN shall be returned.46609 If *x* is +Inf, *x* shall be returned.

46610 If *x* is ±0, a pole error shall occur, and *tgamma()*, *tgammaf()*, and *tgammal()* shall return
 46611 ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL, respectively.

46612 If *x* is −Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-
 46613 defined value shall be returned.

46614 **ERRORS**

46615 These functions shall fail if:

46616 **MX** Domain Error The value of *x* is a negative integer, or *x* is −Inf.

46617 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 46618 then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling
 46619 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 46620 shall be raised.

46621 **MX** Pole Error The value of *x* is zero.

46622 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
 46623 then *errno* shall be set to [ERANGE]. If the integer expression
 46624 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-
 46625 zero floating-point exception shall be raised.

46626 Range Error The value overflows.

46627 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
46628 then *errno* shall be set to [ERANGE]. If the integer expression
46629 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow
46630 floating-point exception shall be raised.

46631 EXAMPLES

46632 None.

46633 APPLICATION USAGE

46634 For IEEE Std 754-1985 **double**, overflow happens when $0 < x < 1/\text{DBL_MAX}$, and $171.7 < x$.

46635 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
46636 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

46637 RATIONALE

46638 This function is named *tgamma()* in order to avoid conflicts with the historical *gamma()* and
46639 *lgamma()* functions.

46640 FUTURE DIRECTIONS

46641 It is possible that the error response for a negative integer argument may be changed to a pole
46642 error and a return value of $\pm\text{Inf}$.

46643 SEE ALSO

46644 *feclearexcept()*, *fetestexcept()*, *lgamma()*, the Base Definitions volume of IEEE Std 1003.1-2001,
46645 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

46646 CHANGE HISTORY

46647 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

46648 **NAME**

46649 time — get time

46650 **SYNOPSIS**

46651 #include <time.h>

46652 time_t time(time_t *tloc);

46653 **DESCRIPTION**

46654 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 46655 conflict between the requirements described here and the ISO C standard is unintentional. This
 46656 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

46657 CX The *time()* function shall return the value of time in seconds since the Epoch.

46658 The *tloc* argument points to an area where the return value is also stored. If *tloc* is a null pointer,
 46659 no value is stored.

46660 **RETURN VALUE**

46661 Upon successful completion, *time()* shall return the value of time. Otherwise, (**time_t**)−1 shall be
 46662 returned.

46663 **ERRORS**

46664 No errors are defined.

46665 **EXAMPLES**46666 **Getting the Current Time**

46667 The following example uses the *time()* function to calculate the time elapsed, in seconds, since
 46668 the Epoch, *localtime()* to convert that value to a broken-down time, and *asctime()* to convert the
 46669 broken-down time values into a printable string.

46670 #include <stdio.h>

46671 #include <time.h>

46672 int main(void)

46673 {

46674 time_t result;

46675 result = time(NULL);

46676 printf("%s%ju secs since the Epoch\n",

46677 asctime(localtime(&result)),

46678 (uintmax_t)result);

46679 return(0);

46680 }

46681 This example writes the current time to *stdout* in a form like this:

46682 Wed Jun 26 10:32:15 1996

46683 835810335 secs since the Epoch

Timing an Event

The following example gets the current time, prints it out in the user's format, and prints the number of minutes to an event being timed.

```
#include <time.h>
#include <stdio.h>
...
time_t now;
int minutes_to_event;
...
time(&now);
minutes_to_event = ...;
printf("The time is ");
puts(asctime(localtime(&now)));
printf("There are %d minutes to the event.\n",
      minutes_to_event);
...
```

APPLICATION USAGE

None.

RATIONALE

The *time()* function returns a value in seconds (type **time_t**) while *times()* returns a set of values in clock ticks (type **clock_t**). Some historical implementations, such as 4.3 BSD, have mechanisms capable of returning more precise times (see below). A generalized timing scheme to unify these various timing mechanisms has been proposed but not adopted.

Implementations in which **time_t** is a 32-bit signed integer (many historical implementations) fail in the year 2038. IEEE Std 1003.1-2001 does not address this problem. However, the use of the **time_t** type is mandated in order to ease the eventual fix.

The use of the **<time.h>** header instead of **<sys/types.h>** allows compatibility with the ISO C standard.

Many historical implementations (including Version 7) and the 1984 /usr/group standard use **long** instead of **time_t**. This volume of IEEE Std 1003.1-2001 uses the latter type in order to agree with the ISO C standard.

4.3 BSD includes *time()* only as an alternate function to the more flexible *gettimeofday()* function.

FUTURE DIRECTIONS

In a future version of this volume of IEEE Std 1003.1-2001, **time_t** is likely to be required to be capable of representing times far in the future. Whether this will be mandated as a 64-bit type or a requirement that a specific date in the future be representable (for example, 10000 AD) is not yet determined. Systems purchased after the approval of this volume of IEEE Std 1003.1-2001 should be evaluated to determine whether their lifetime will extend past 2038.

SEE ALSO

asctime(), *clock()*, *ctime()*, *difftime()*, *gettimeofday()*, *gmtime()*, *localtime()*, *mktime()*, *strftime()*, *strptime()*, *utime()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<time.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

46727 **Issue 6**

46728 Extensions beyond the ISO C standard are marked.

46729 The EXAMPLES, RATIONALE, and FUTURE DIRECTIONS sections are added.

46730 NAME

46731 timer_create — create a per-process timer (**REALTIME**)

46732 SYNOPSIS

```
46733 TMR    #include <signal.h>
46734        #include <time.h>
```

```
46735        int timer_create(clockid_t clockid, struct sigevent *restrict evp,
46736                        timer_t *restrict timerid);
46737
```

46738 DESCRIPTION

46739 The *timer_create()* function shall create a per-process timer using the specified clock, *clock_id*, as
 46740 the timing base. The *timer_create()* function shall return, in the location referenced by *timerid*, a
 46741 timer ID of type **timer_t** used to identify the timer in timer requests. This timer ID shall be
 46742 unique within the calling process until the timer is deleted. The particular clock, *clock_id*, is
 46743 defined in **<time.h>**. The timer whose ID is returned shall be in a disarmed state upon return
 46744 from *timer_create()*.

46745 The *evp* argument, if non-NULL, points to a **sigevent** structure. This structure, allocated by the
 46746 application, defines the asynchronous notification to occur as specified in Section 2.4.1 (on page
 46747 28) when the timer expires. If the *evp* argument is NULL, the effect is as if the *evp* argument
 46748 pointed to a **sigevent** structure with the *sigev_notify* member having the value SIGEV_SIGNAL,
 46749 the *sigev_signo* having a default signal number, and the *sigev_value* member having the value of
 46750 the timer ID.

46751 Each implementation shall define a set of clocks that can be used as timing bases for per-process
 46752 MON timers. All implementations shall support a *clock_id* of CLOCK_REALTIME. If the Monotonic
 46753 Clock option is supported, implementations shall support a *clock_id* of CLOCK_MONOTONIC.

46754 Per-process timers shall not be inherited by a child process across a *fork()* and shall be disarmed
 46755 and deleted by an *exec*.

46756 CPT If _POSIX_CPUTIME is defined, implementations shall support *clock_id* values representing the
 46757 CPU-time clock of the calling process.

46758 TCT If _POSIX_THREAD_CPUTIME is defined, implementations shall support *clock_id* values
 46759 representing the CPU-time clock of the calling thread.

46760 CPT|TCT It is implementation-defined whether a *timer_create()* function will succeed if the value defined
 46761 by *clock_id* corresponds to the CPU-time clock of a process or thread different from the process
 46762 or thread invoking the function.

46763 RETURN VALUE

46764 If the call succeeds, *timer_create()* shall return zero and update the location referenced by *timerid*
 46765 to a **timer_t**, which can be passed to the per-process timer calls. If an error occurs, the function
 46766 shall return a value of -1 and set *errno* to indicate the error. The value of *timerid* is undefined if
 46767 an error occurs.

46768 ERRORS

46769 The *timer_create()* function shall fail if:

- | | | |
|-------|----------|---|
| 46770 | [EAGAIN] | The system lacks sufficient signal queuing resources to honor the request. |
| 46771 | [EAGAIN] | The calling process has already created all of the timers it is allowed by this |
| 46772 | | implementation. |
| 46773 | [EINVAL] | The specified clock ID is not defined. |

46774 CPT|TCT [ENOTSUP] The implementation does not support the creation of a timer attached to the
 46775 CPU-time clock that is specified by *clock_id* and associated with a process or
 46776 thread different from the process or thread invoking *timer_create()*.

46777 EXAMPLES

46778 None.

46779 APPLICATION USAGE

46780 None.

46781 RATIONALE

46782 Periodic Timer Overrun and Resource Allocation

46783 The specified timer facilities may deliver realtime signals (that is, queued signals) on
 46784 implementations that support this option. Since realtime applications cannot afford to lose
 46785 notifications of asynchronous events, like timer expirations or asynchronous I/O completions, it
 46786 must be possible to ensure that sufficient resources exist to deliver the signal when the event
 46787 occurs. In general, this is not a difficulty because there is a one-to-one correspondence between a
 46788 request and a subsequent signal generation. If the request cannot allocate the signal delivery
 46789 resources, it can fail the call with an [EAGAIN] error.

46790 Periodic timers are a special case. A single request can generate an unspecified number of
 46791 signals. This is not a problem if the requesting process can service the signals as fast as they are
 46792 generated, thus making the signal delivery resources available for delivery of subsequent
 46793 periodic timer expiration signals. But, in general, this cannot be assured—processing of periodic
 46794 timer signals may “overrun”; that is, subsequent periodic timer expirations may occur before the
 46795 currently pending signal has been delivered.

46796 Also, for signals, according to the POSIX.1-1990 standard, if subsequent occurrences of a
 46797 pending signal are generated, it is implementation-defined whether a signal is delivered for each
 46798 occurrence. This is not adequate for some realtime applications. So a mechanism is required to
 46799 allow applications to detect how many timer expirations were delayed without requiring an
 46800 indefinite amount of system resources to store the delayed expirations.

46801 The specified facilities provide for an overrun count. The overrun count is defined as the number
 46802 of extra timer expirations that occurred between the time a timer expiration signal is generated
 46803 and the time the signal is delivered. The signal-catching function, if it is concerned with
 46804 overruns, can retrieve this count on entry. With this method, a periodic timer only needs one
 46805 “signal queuing resource” that can be allocated at the time of the *timer_create()* function call.

46806 A function is defined to retrieve the overrun count so that an application need not allocate static
 46807 storage to contain the count, and an implementation need not update this storage
 46808 asynchronously on timer expirations. But, for some high-frequency periodic applications, the
 46809 overhead of an additional system call on each timer expiration may be prohibitive. The
 46810 functions, as defined, permit an implementation to maintain the overrun count in user space,
 46811 associated with the *timerid*. The *timer_getoverrun()* function can then be implemented as a macro
 46812 that uses the *timerid* argument (which may just be a pointer to a user space structure containing
 46813 the counter) to locate the overrun count with no system call overhead. Other implementations,
 46814 less concerned with this class of applications, can avoid the asynchronous update of user space
 46815 by maintaining the count in a system structure at the cost of the extra system call to obtain it.

46816 **Timer Expiration Signal Parameters**

46817 The Realtime Signals Extension option supports an application-specific datum that is delivered
46818 to the extended signal handler. This value is explicitly specified by the application, along with
46819 the signal number to be delivered, in a **sigevent** structure. The type of the application-defined
46820 value can be either an integer constant or a pointer. This explicit specification of the value, as
46821 opposed to always sending the timer ID, was selected based on existing practice.

46822 It is common practice for realtime applications (on non-POSIX systems or realtime extended
46823 POSIX systems) to use the parameters of event handlers as the case label of a switch statement
46824 or as a pointer to an application-defined data structure. Since *timer_ids* are dynamically allocated
46825 by the *timer_create()* function, they can be used for neither of these functions without additional
46826 application overhead in the signal handler; for example, to search an array of saved timer IDs to
46827 associate the ID with a constant or application data structure.

46828 **FUTURE DIRECTIONS**

46829 None.

46830 **SEE ALSO**

46831 *clock_getres()*, *timer_delete()*, *timer_getoverrun()*, the Base Definitions volume of
46832 IEEE Std 1003.1-2001, <**time.h**>

46833 **CHANGE HISTORY**

46834 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

46835 **Issue 6**

46836 The *timer_create()* function is marked as part of the Timers option.

46837 The [ENOSYS] error condition has been removed as stubs need not be provided if an
46838 implementation does not support the Timers option.

46839 CPU-time clocks are added for alignment with IEEE Std 1003.1d-1999.

46840 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding the
46841 requirement for the CLOCK_MONOTONIC clock under the Monotonic Clock option.

46842 The **restrict** keyword is added to the *timer_create()* prototype for alignment with the
46843 ISO/IEC 9899:1999 standard.

46844 **NAME**46845 timer_delete — delete a per-process timer (**REALTIME**)46846 **SYNOPSIS**46847 TMR `#include <time.h>`46848 `int timer_delete(timer_t timerid);`

46849

46850 **DESCRIPTION**

46851 The *timer_delete()* function deletes the specified timer, *timerid*, previously created by the
46852 *timer_create()* function. If the timer is armed when *timer_delete()* is called, the behavior shall be
46853 as if the timer is automatically disarmed before removal. The disposition of pending signals for
46854 the deleted timer is unspecified.

46855 **RETURN VALUE**

46856 If successful, the *timer_delete()* function shall return a value of zero. Otherwise, the function shall
46857 return a value of `-1` and set *errno* to indicate the error.

46858 **ERRORS**46859 The *timer_delete()* function shall fail if:46860 [EINVAL] The timer ID specified by *timerid* is not a valid timer ID.46861 **EXAMPLES**

46862 None.

46863 **APPLICATION USAGE**

46864 None.

46865 **RATIONALE**

46866 None.

46867 **FUTURE DIRECTIONS**

46868 None.

46869 **SEE ALSO**46870 *timer_create()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<time.h>**46871 **CHANGE HISTORY**

46872 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

46873 **Issue 6**46874 The *timer_delete()* function is marked as part of the Timers option.

46875 The [ENOSYS] error condition has been removed as stubs need not be provided if an
46876 implementation does not support the Timers option.

46877 NAME

46878 timer_getoverrun, timer_gettime, timer_settime — per-process timers (**REALTIME**)

46879 SYNOPSIS

46880 TMR `#include <time.h>`

```

46881 int timer_getoverrun(timer_t timerid);
46882 int timer_gettime(timer_t timerid, struct itimerspec *value);
46883 int timer_settime(timer_t timerid, int flags,
46884     const struct itimerspec *restrict value,
46885     struct itimerspec *restrict ovalue);
46886

```

46887 DESCRIPTION

46888 The *timer_gettime()* function shall store the amount of time until the specified timer, *timerid*,
 46889 expires and the reload value of the timer into the space pointed to by the *value* argument. The
 46890 *it_value* member of this structure shall contain the amount of time before the timer expires, or
 46891 zero if the timer is disarmed. This value is returned as the interval until timer expiration, even if
 46892 the timer was armed with absolute time. The *it_interval* member of *value* shall contain the reload
 46893 value last set by *timer_settime()*.

46894 The *timer_settime()* function shall set the time until the next expiration of the timer specified by
 46895 *timerid* from the *it_value* member of the *value* argument and arm the timer if the *it_value* member
 46896 of *value* is non-zero. If the specified timer was already armed when *timer_settime()* is called, this
 46897 call shall reset the time until next expiration to the *value* specified. If the *it_value* member of *value*
 46898 is zero, the timer shall be disarmed. The effect of disarming or resetting a timer with pending
 46899 expiration notifications is unspecified.

46900 If the flag **TIMER_ABSTIME** is not set in the argument *flags*, *timer_settime()* shall behave as if the
 46901 time until next expiration is set to be equal to the interval specified by the *it_value* member of
 46902 *value*. That is, the timer shall expire in *it_value* nanoseconds from when the call is made. If the
 46903 flag **TIMER_ABSTIME** is set in the argument *flags*, *timer_settime()* shall behave as if the time
 46904 until next expiration is set to be equal to the difference between the absolute time specified by
 46905 the *it_value* member of *value* and the current value of the clock associated with *timerid*. That is,
 46906 the timer shall expire when the clock reaches the value specified by the *it_value* member of *value*.
 46907 If the specified time has already passed, the function shall succeed and the expiration
 46908 notification shall be made.

46909 The reload value of the timer shall be set to the value specified by the *it_interval* member of
 46910 *value*. When a timer is armed with a non-zero *it_interval*, a periodic (or repetitive) timer is
 46911 specified.

46912 Time values that are between two consecutive non-negative integer multiples of the resolution
 46913 of the specified timer shall be rounded up to the larger multiple of the resolution. Quantization
 46914 error shall not cause the timer to expire earlier than the rounded time value.

46915 If the argument *ovalue* is not NULL, the *timer_settime()* function shall store, in the location
 46916 referenced by *ovalue*, a value representing the previous amount of time before the timer would
 46917 have expired, or zero if the timer was disarmed, together with the previous timer reload value.
 46918 Timers shall not expire before their scheduled time.

46919 Only a single signal shall be queued to the process for a given timer at any point in time. When a
 46920 timer for which a signal is still pending expires, no signal shall be queued, and a timer overrun
 46921 shall occur. When a timer expiration signal is delivered to or accepted by a process, if the
 46922 implementation supports the Realtime Signals Extension, the *timer_getoverrun()* function shall
 46923 return the timer expiration overrun count for the specified timer. The overrun count returned
 46924 contains the number of extra timer expirations that occurred between the time the signal was

generated (queued) and when it was delivered or accepted, up to but not including an implementation-defined maximum of {DELAYTIMER_MAX}. If the number of such extra expirations is greater than or equal to {DELAYTIMER_MAX}, then the overrun count shall be set to {DELAYTIMER_MAX}. The value returned by *timer_getoverrun()* shall apply to the most recent expiration signal delivery or acceptance for the timer. If no expiration signal has been delivered for the timer, or if the Realtime Signals Extension is not supported, the return value of *timer_getoverrun()* is unspecified.

46932 RETURN VALUE

46933 If the *timer_getoverrun()* function succeeds, it shall return the timer expiration overrun count as explained above.

46935 If the *timer_gettime()* or *timer_settime()* functions succeed, a value of 0 shall be returned.

46936 If an error occurs for any of these functions, the value -1 shall be returned, and *errno* set to indicate the error.

46938 ERRORS

46939 The *timer_getoverrun()*, *timer_gettime()*, and *timer_settime()* functions shall fail if:

46940 [EINVAL] The *timerid* argument does not correspond to an ID returned by *timer_create()*
46941 but not yet deleted by *timer_delete()*.

46942 The *timer_settime()* function shall fail if:

46943 [EINVAL] A *value* structure specified a nanosecond value less than zero or greater than
46944 or equal to 1 000 million, and the *it_value* member of that structure did not
46945 specify zero seconds and nanoseconds.

46946 EXAMPLES

46947 None.

46948 APPLICATION USAGE

46949 None.

46950 RATIONALE

46951 Practical clocks tick at a finite rate, with rates of 100 hertz and 1 000 hertz being common. The
46952 inverse of this tick rate is the clock resolution, also called the clock granularity, which in either
46953 case is expressed as a time duration, being 10 milliseconds and 1 millisecond respectively for
46954 these common rates. The granularity of practical clocks implies that if one reads a given clock
46955 twice in rapid succession, one may get the same time value twice; and that timers must wait for
46956 the next clock tick after the theoretical expiration time, to ensure that a timer never returns too
46957 soon. Note also that the granularity of the clock may be significantly coarser than the resolution
46958 of the data format used to set and get time and interval values. Also note that some
46959 implementations may choose to adjust time and/or interval values to exactly match the ticks of
46960 the underlying clock.

46961 This volume of IEEE Std 1003.1-2001 defines functions that allow an application to determine the
46962 implementation-supported resolution for the clocks and requires an implementation to
46963 document the resolution supported for timers and *nanosleep()* if they differ from the supported
46964 clock resolution. This is more of a procurement issue than a runtime application issue.

46965 FUTURE DIRECTIONS

46966 None.

46967 **SEE ALSO**

46968 *clock_getres()*, *timer_create()*, the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>

46969 **CHANGE HISTORY**

46970 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

46971 **Issue 6**

46972 The *timer_getoverrun()*, *timer_gettime()*, and *timer_settime()* functions are marked as part of the
46973 Timers option.

46974 The [ENOSYS] error condition has been removed as stubs need not be provided if an
46975 implementation does not support the Timers option.

46976 The [EINVAL] error condition is updated to include the following: “and the *it_value* member of
46977 that structure did not specify zero seconds and nanoseconds.” This change is for IEEE PASC
46978 Interpretation 1003.1 #89.

46979 The DESCRIPTION for *timer_getoverrun()* is updated to clarify that “If no expiration signal has
46980 been delivered for the timer, or if the Realtime Signals Extension is not supported, the return
46981 value of *timer_getoverrun()* is unspecified”.

46982 The **restrict** keyword is added to the *timer_settime()* prototype for alignment with the
46983 ISO/IEC 9899:1999 standard.

46984 **NAME**

46985 times — get process and waited-for child process times

46986 **SYNOPSIS**

46987 #include <sys/times.h>

46988 clock_t times(struct tms *buffer);

46989 **DESCRIPTION**46990 The *times()* function shall fill the **tms** structure pointed to by *buffer* with time-accounting information. The **tms** structure is defined in <sys/times.h>.

46992 All times are measured in terms of the number of clock ticks used.

46993 The times of a terminated child process shall be included in the *tms_cutime* and *tms_cstime* elements of the parent when *wait()* or *waitpid()* returns the process ID of this terminated child. If a child process has not waited for its children, their times shall not be included in its times.46996 • The *tms_utime* structure member is the CPU time charged for the execution of user instructions of the calling process.46998 • The *tms_stime* structure member is the CPU time charged for execution by the system on behalf of the calling process.47000 • The *tms_cutime* structure member is the sum of the *tms_utime* and *tms_cutime* times of the child processes.47002 • The *tms_cstime* structure member is the sum of the *tms_stime* and *tms_cstime* times of the child processes.47004 **RETURN VALUE**47005 Upon successful completion, *times()* shall return the elapsed real time, in clock ticks, since an arbitrary point in the past (for example, system start-up time). This point does not change from one invocation of *times()* within the process to another. The return value may overflow the possible range of type **clock_t**. If *times()* fails, (**clock_t**)-1 shall be returned and *errno* set to indicate the error.47010 **ERRORS**

47011 No errors are defined.

47012 **EXAMPLES**47013 **Timing a Database Lookup**47014 The following example defines two functions, *start_clock()* and *end_clock()*, that are used to time a lookup. It also defines variables of type **clock_t** and **tms** to measure the duration of transactions. The *start_clock()* function saves the beginning times given by the *times()* function. The *end_clock()* function gets the ending times and prints the difference between the two times.

```

47018     #include <sys/times.h>
47019     #include <stdio.h>
47020     ...
47021     void start_clock(void);
47022     void end_clock(char *msg);
47023     ...
47024     static clock_t st_time;
47025     static clock_t en_time;
47026     static struct tms st_cpu;
47027     static struct tms en_cpu;

```



```

47028     ...
47029     void
47030     start_clock()
47031     {
47032         st_time = times(&st_cpu);
47033     }

47034     /* This example assumes that the result of each subtraction
47035        is within the range of values that can be represented in
47036        an integer type. */
47037     void
47038     end_clock(char *msg)
47039     {
47040         en_time = times(&en_cpu);

47041         fputs(msg, stdout);
47042         printf("Real Time: %jd, User Time %jd, System Time %jd\n",
47043             (intmax_t)(en_time - st_time),
47044             (intmax_t)(en_cpu.tms_utime - st_cpu.tms_utime),
47045             (intmax_t)(en_cpu.tms_stime - st_cpu.tms_stime));
47046     }

```

47047 APPLICATION USAGE

47048 Applications should use `sysconf(_SC_CLK_TCK)` to determine the number of clock ticks per
 47049 second as it may vary from system to system.

47050 RATIONALE

47051 The accuracy of the times reported is intentionally left unspecified to allow implementations
 47052 flexibility in design, from uniprocessor to multi-processor networks.

47053 The inclusion of times of child processes is recursive, so that a parent process may collect the
 47054 total times of all of its descendants. But the times of a child are only added to those of its parent
 47055 when its parent successfully waits on the child. Thus, it is not guaranteed that a parent process
 47056 can always see the total times of all its descendants; see also the discussion of the term
 47057 “realtime” in *alarm()*.

47058 If the type `clock_t` is defined to be a signed 32-bit integer, it overflows in somewhat more than a
 47059 year if there are 60 clock ticks per second, or less than a year if there are 100. There are individual
 47060 systems that run continuously for longer than that. This volume of IEEE Std 1003.1-2001 permits
 47061 an implementation to make the reference point for the returned value be the start-up time of the
 47062 process, rather than system start-up time.

47063 The term “charge” in this context has nothing to do with billing for services. The operating
 47064 system accounts for time used in this way. That information must be correct, regardless of how
 47065 that information is used.

47066 FUTURE DIRECTIONS

47067 None.

47068 SEE ALSO

47069 *alarm()*, *exec*, *fork()*, *sysconf()*, *time()*, *wait()*, the Base Definitions volume of
 47070 IEEE Std 1003.1-2001, <sys/times.h>

47071 CHANGE HISTORY

47072 First released in Issue 1. Derived from Issue 1 of the SVID.

47073 **NAME**

47074 timezone — difference from UTC and local standard time

47075 **SYNOPSIS**

47076 xSI #include <time.h>

47077 extern long timezone;

47078

47079 **DESCRIPTION**47080 Refer to *tzset()*.

47081 **NAME**

47082 tmpfile — create a temporary file

47083 **SYNOPSIS**

47084 #include <stdio.h>

47085 FILE *tmpfile(void);

47086 **DESCRIPTION**

47087 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 47088 conflict between the requirements described here and the ISO C standard is unintentional. This
 47089 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

47090 The *tmpfile()* function shall create a temporary file and open a corresponding stream. The file
 47091 shall be automatically deleted when all references to the file are closed. The file is opened as in
 47092 *fopen()* for update (w+).

47093 CX In some implementations, a permanent file may be left behind if the process calling *tmpfile()* is
 47094 killed while it is processing a call to *tmpfile()*.

47095 An error message may be written to standard error if the stream cannot be opened.

47096 **RETURN VALUE**

47097 Upon successful completion, *tmpfile()* shall return a pointer to the stream of the file that is
 47098 CX created. Otherwise, it shall return a null pointer and set *errno* to indicate the error.

47099 **ERRORS**47100 The *tmpfile()* function shall fail if:

47101 CX [EINTR] A signal was caught during *tmpfile()*.

47102 CX [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

47103 CX [ENFILE] The maximum allowable number of files is currently open in the system.

47104 CX [ENOSPC] The directory or file system which would contain the new file cannot be
 47105 expanded.

47106 CX [EOVERFLOW] The file is a regular file and the size of the file cannot be represented correctly
 47107 in an object of type *off_t*.

47108 The *tmpfile()* function may fail if:

47109 CX [EMFILE] {FOPEN_MAX} streams are currently open in the calling process.

47110 CX [ENOMEM] Insufficient storage space is available.

47111 **EXAMPLES**47112 **Creating a Temporary File**

47113 The following example creates a temporary file for update, and returns a pointer to a stream for
 47114 the created file in the *fp* variable.

47115 #include <stdio.h>

47116 ...

47117 FILE *fp;

47118 fp = tmpfile ();

47119 APPLICATION USAGE

47120 It should be possible to open at least {TMP_MAX} temporary files during the lifetime of the
47121 program (this limit may be shared with *tmpnam()*) and there should be no limit on the number
47122 simultaneously open other than this limit and any limit on the number of open files
47123 ({FOPEN_MAX}).

47124 RATIONALE

47125 None.

47126 FUTURE DIRECTIONS

47127 None.

47128 SEE ALSO

47129 *fopen()*, *tmpnam()*, *unlink()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

47130 CHANGE HISTORY

47131 First released in Issue 1. Derived from Issue 1 of the SVID.

47132 Issue 5

47133 Large File Summit extensions are added.

47134 The last two paragraphs of the DESCRIPTION were included as APPLICATION USAGE notes
47135 in previous issues.

47136 Issue 6

47137 Extensions beyond the ISO C standard are marked.

47138 The following new requirements on POSIX implementations derive from alignment with the
47139 Single UNIX Specification:

- 47140 • In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support
47141 large files.
- 47142 • The [EMFILE] optional error condition is added.

47143 The APPLICATION USAGE section is added for alignment with the ISO/IEC 9899:1999
47144 standard.

47145 **NAME**

47146 tmpnam — create a name for a temporary file

47147 **SYNOPSIS**

47148 #include <stdio.h>

47149 char *tmpnam(char *s);

47150 **DESCRIPTION**

47151 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 47152 conflict between the requirements described here and the ISO C standard is unintentional. This
 47153 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

47154 The *tmpnam()* function shall generate a string that is a valid filename and that is not the same as
 47155 the name of an existing file. The function is potentially capable of generating {TMP_MAX}
 47156 different strings, but any or all of them may already be in use by existing files and thus not be
 47157 suitable return values.

47158 The *tmpnam()* function generates a different string each time it is called from the same process,
 47159 up to {TMP_MAX} times. If it is called more than {TMP_MAX} times, the behavior is
 47160 implementation-defined.

47161 The implementation shall behave as if no function defined in this volume of
 47162 IEEE Std 1003.1-2001 calls *tmpnam()*.

47163 CX If the application uses any of the functions guaranteed to be available if either
 47164 _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS is defined, the application shall
 47165 ensure that the *tmpnam()* function is called with a non-NULL parameter.

47166 **RETURN VALUE**

47167 Upon successful completion, *tmpnam()* shall return a pointer to a string. If no suitable string can
 47168 be generated, the *tmpnam()* function shall return a null pointer.

47169 If the argument *s* is a null pointer, *tmpnam()* shall leave its result in an internal static object and
 47170 return a pointer to that object. Subsequent calls to *tmpnam()* may modify the same object. If the
 47171 argument *s* is not a null pointer, it is presumed to point to an array of at least L_tmpnam chars;
 47172 *tmpnam()* shall write its result in that array and shall return the argument as its value.

47173 **ERRORS**

47174 No errors are defined.

47175 **EXAMPLES**47176 **Generating a Filename**47177 The following example generates a unique filename and stores it in the array pointed to by *ptr*.

47178 #include <stdio.h>

47179 ...

47180 char filename[L_tmpnam+1];

47181 char *ptr;

47182 ptr = tmpnam(filename);

47183 **APPLICATION USAGE**

47184 This function only creates filenames. It is the application's responsibility to create and remove
 47185 the files.

47186 Between the time a pathname is created and the file is opened, it is possible for some other
 47187 process to create a file with the same name. Applications may find *tmpfile()* more useful.

47188 **RATIONALE**

47189 None.

47190 **FUTURE DIRECTIONS**

47191 None.

47192 **SEE ALSO**

47193 *fopen()*, *open()*, *tmpnam()*, *tmpfile()*, *unlink()*, the Base Definitions volume of
47194 IEEE Std 1003.1-2001, <stdio.h>

47195 **CHANGE HISTORY**

47196 First released in Issue 1. Derived from Issue 1 of the SVID.

47197 **Issue 5**

47198 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

47199 **Issue 6**

47200 Extensions beyond the ISO C standard are marked.

47201 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

47202 The DESCRIPTION is expanded for alignment with the ISO/IEC 9899:1999 standard.

47203 **NAME**

47204 toascii — translate an integer to a 7-bit ASCII character

47205 **SYNOPSIS**

47206 xSI #include <ctype.h>

47207 int toascii(int c);

47208

47209 **DESCRIPTION**47210 The *toascii()* function shall convert its argument into a 7-bit ASCII character.47211 **RETURN VALUE**47212 The *toascii()* function shall return the value (*c* & 0x7f).47213 **ERRORS**

47214 No errors are returned.

47215 **EXAMPLES**

47216 None.

47217 **APPLICATION USAGE**

47218 None.

47219 **RATIONALE**

47220 None.

47221 **FUTURE DIRECTIONS**

47222 None.

47223 **SEE ALSO**47224 *isascii()*, the Base Definitions volume of IEEE Std 1003.1-2001, <ctype.h>47225 **CHANGE HISTORY**

47226 First released in Issue 1. Derived from Issue 1 of the SVID.

47227 **NAME**

47228 tolower — transliterate uppercase characters to lowercase

47229 **SYNOPSIS**

47230 #include <ctype.h>

47231 int tolower(int c);

47232 **DESCRIPTION**

47233 CX The functionality described on this reference page is aligned with the ISO C standard. Any
47234 conflict between the requirements described here and the ISO C standard is unintentional. This
47235 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

47236 The *tolower()* function has as a domain a type **int**, the value of which is representable as an
47237 **unsigned char** or the value of EOF. If the argument has any other value, the behavior is
47238 undefined. If the argument of *tolower()* represents an uppercase letter, and there exists a
47239 CX corresponding lowercase letter (as defined by character type information in the program locale
47240 category *LC_CTYPE*), the result shall be the corresponding lowercase letter. All other arguments
47241 in the domain are returned unchanged.

47242 **RETURN VALUE**

47243 Upon successful completion, *tolower()* shall return the lowercase letter corresponding to the
47244 argument passed; otherwise, it shall return the argument unchanged.

47245 **ERRORS**

47246 No errors are defined.

47247 **EXAMPLES**

47248 None.

47249 **APPLICATION USAGE**

47250 None.

47251 **RATIONALE**

47252 None.

47253 **FUTURE DIRECTIONS**

47254 None.

47255 **SEE ALSO**47256 *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <ctype.h>47257 **CHANGE HISTORY**

47258 First released in Issue 1. Derived from Issue 1 of the SVID.

47259 **Issue 6**

47260 Extensions beyond the ISO C standard are marked.

47261 **NAME**

47262 toupper — transliterate lowercase characters to uppercase

47263 **SYNOPSIS**

47264 #include <ctype.h>

47265 int toupper(int c);

47266 **DESCRIPTION**

47267 CX The functionality described on this reference page is aligned with the ISO C standard. Any
47268 conflict between the requirements described here and the ISO C standard is unintentional. This
47269 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

47270 The *toupper()* function has as a domain a type **int**, the value of which is representable as an
47271 **unsigned char** or the value of EOF. If the argument has any other value, the behavior is
47272 undefined. If the argument of *toupper()* represents a lowercase letter, and there exists a
47273 CX corresponding uppercase letter (as defined by character type information in the program locale
47274 category *LC_CTYPE*), the result shall be the corresponding uppercase letter. All other arguments
47275 in the domain are returned unchanged.

47276 **RETURN VALUE**

47277 Upon successful completion, *toupper()* shall return the uppercase letter corresponding to the
47278 argument passed.

47279 **ERRORS**

47280 No errors are defined.

47281 **EXAMPLES**

47282 None.

47283 **APPLICATION USAGE**

47284 None.

47285 **RATIONALE**

47286 None.

47287 **FUTURE DIRECTIONS**

47288 None.

47289 **SEE ALSO**47290 *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <ctype.h>47291 **CHANGE HISTORY**

47292 First released in Issue 1. Derived from Issue 1 of the SVID.

47293 **Issue 6**

47294 Extensions beyond the ISO C standard are marked.

47295 **NAME**

47296 towctrans — wide-character transliteration

47297 **SYNOPSIS**

47298 #include <wctype.h>

47299 wint_t towctrans(wint_t *wc*, wctrans_t *desc*);47300 **DESCRIPTION**

47301 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 47302 conflict between the requirements described here and the ISO C standard is unintentional. This
 47303 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

47304 The *towctrans()* function shall transliterate the wide-character code *wc* using the mapping
 47305 described by *desc*. The current setting of the *LC_CTYPE* category should be the same as during
 47306 CX the call to *wctrans()* that returned the value *desc*. If the value of *desc* is invalid (that is, not
 47307 obtained by a call to *wctrans()* or *desc* is invalidated by a subsequent call to *setlocale()* that has
 47308 affected category *LC_CTYPE*), the result is unspecified.

47309 An application wishing to check for error situations should set *errno* to 0 before calling
 47310 *towctrans()*. If *errno* is non-zero on return, an error has occurred.

47311 **RETURN VALUE**

47312 If successful, the *towctrans()* function shall return the mapped value of *wc* using the mapping
 47313 described by *desc*. Otherwise, it shall return *wc* unchanged.

47314 **ERRORS**47315 The *towctrans()* function may fail if:47316 CX [EINVAL] *desc* contains an invalid transliteration descriptor.47317 **EXAMPLES**

47318 None.

47319 **APPLICATION USAGE**

47320 The strings "tolower" and "toupper" are reserved for the standard mapping names. In the
 47321 table below, the functions in the left column are equivalent to the functions in the right column.

47322 tolower(<i>wc</i>)	towctrans(<i>wc</i> , wctrans("tolower"))
47323 toupper(<i>wc</i>)	towctrans(<i>wc</i> , wctrans("toupper"))

47324 **RATIONALE**

47325 None.

47326 **FUTURE DIRECTIONS**

47327 None.

47328 **SEE ALSO**

47329 *tolower()*, *toupper()*, *wctrans()*, the Base Definitions volume of IEEE Std 1003.1-2001,
 47330 <wctype.h>

47331 **CHANGE HISTORY**

47332 First released in Issue 5. Derived from ISO/IEC 9899:1990/Amendment 1:1995 (E).

47333 **Issue 6**

47334 Extensions beyond the ISO C standard are marked.

47335 NAME

47336 towlower — transliterate uppercase wide-character code to lowercase

47337 SYNOPSIS

47338 #include <wctype.h>

47339 wint_t towlower(wint_t wc);

47340 DESCRIPTION

47341 cx The functionality described on this reference page is aligned with the ISO C standard. Any
47342 conflict between the requirements described here and the ISO C standard is unintentional. This
47343 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

47344 The *towlower()* function has as a domain a type **wint_t**, the value of which the application shall
47345 ensure is a character representable as a **wchar_t**, and a wide-character code corresponding to a
47346 valid character in the current locale or the value of WEOF. If the argument has any other value,
47347 the behavior is undefined. If the argument of *towlower()* represents an uppercase wide-character
47348 code, and there exists a corresponding lowercase wide-character code (as defined by character
47349 type information in the program locale category *LC_CTYPE*), the result shall be the
47350 corresponding lowercase wide-character code. All other arguments in the domain are returned
47351 unchanged.

47352 RETURN VALUE

47353 Upon successful completion, *towlower()* shall return the lowercase letter corresponding to the
47354 argument passed; otherwise, it shall return the argument unchanged.

47355 ERRORS

47356 No errors are defined.

47357 EXAMPLES

47358 None.

47359 APPLICATION USAGE

47360 None.

47361 RATIONALE

47362 None.

47363 FUTURE DIRECTIONS

47364 None.

47365 SEE ALSO

47366 *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <**wctype.h**>,
47367 <**wchar.h**>

47368 CHANGE HISTORY

47369 First released in Issue 4.

47370 Issue 5

47371 The following change has been made in this issue for alignment with
47372 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 47373 • The SYNOPSIS has been changed to indicate that this function and associated data types are
47374 now made visible by inclusion of the <**wctype.h**> header rather than <**wchar.h**>.

47375 Issue 6

47376 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

47377 **NAME**

47378 towupper — transliterate lowercase wide-character code to uppercase

47379 **SYNOPSIS**

47380 #include <wctype.h>

47381 wint_t towupper(wint_t wc);

47382 **DESCRIPTION**

47383 cx The functionality described on this reference page is aligned with the ISO C standard. Any
 47384 conflict between the requirements described here and the ISO C standard is unintentional. This
 47385 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

47386 The *towupper()* function has as a domain a type **wint_t**, the value of which the application shall
 47387 ensure is a character representable as a **wchar_t**, and a wide-character code corresponding to a
 47388 valid character in the current locale or the value of WEOF. If the argument has any other value,
 47389 the behavior is undefined. If the argument of *towupper()* represents a lowercase wide-character
 47390 code, and there exists a corresponding uppercase wide-character code (as defined by character
 47391 type information in the program locale category *LC_CTYPE*), the result shall be the
 47392 corresponding uppercase wide-character code. All other arguments in the domain are returned
 47393 unchanged.

47394 **RETURN VALUE**

47395 Upon successful completion, *towupper()* shall return the uppercase letter corresponding to the
 47396 argument passed. Otherwise, it shall return the argument unchanged.

47397 **ERRORS**

47398 No errors are defined.

47399 **EXAMPLES**

47400 None.

47401 **APPLICATION USAGE**

47402 None.

47403 **RATIONALE**

47404 None.

47405 **FUTURE DIRECTIONS**

47406 None.

47407 **SEE ALSO**

47408 *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <wctype.h>,
 47409 <wchar.h>

47410 **CHANGE HISTORY**

47411 First released in Issue 4.

47412 **Issue 5**

47413 The following change has been made in this issue for alignment with
 47414 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 47415 • The SYNOPSIS has been changed to indicate that this function and associated data types are
 47416 now made visible by inclusion of the <wctype.h> header rather than <wchar.h>.

47417 **Issue 6**

47418 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

47419 **NAME**

47420 trunc, truncf, trunc1 — round to truncated integer value

47421 **SYNOPSIS**

47422 #include <math.h>

47423 double trunc(double x);

47424 float truncf(float x);

47425 long double trunc1(long double x);

47426 **DESCRIPTION**

47427 CX The functionality described on this reference page is aligned with the ISO C standard. Any
47428 conflict between the requirements described here and the ISO C standard is unintentional. This
47429 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

47430 These functions shall round their argument to the integer value, in floating format, nearest to but
47431 no larger in magnitude than the argument.

47432 **RETURN VALUE**

47433 Upon successful completion, these functions shall return the truncated integer value.

47434 MX If x is NaN, a NaN shall be returned.47435 If x is ± 0 or $\pm \text{Inf}$, x shall be returned.47436 **ERRORS**

47437 No errors are defined.

47438 **EXAMPLES**

47439 None.

47440 **APPLICATION USAGE**

47441 None.

47442 **RATIONALE**

47443 None.

47444 **FUTURE DIRECTIONS**

47445 None.

47446 **SEE ALSO**

47447 The Base Definitions volume of IEEE Std 1003.1-2001, <math.h>

47448 **CHANGE HISTORY**

47449 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

47450 **NAME**

47451 truncate — truncate a file to a specified length

47452 **SYNOPSIS**47453 XSI `#include <unistd.h>`47454 `int truncate(const char *path, off_t length);`

47455

47456 **DESCRIPTION**47457 The *truncate()* function shall cause the regular file named by *path* to have a size which shall be
47458 equal to *length* bytes.47459 If the file previously was larger than *length*, the extra data is discarded. If the file was previously
47460 shorter than *length*, its size is increased, and the extended area appears as if it were zero-filled.

47461 The application shall ensure that the process has write permission for the file.

47462 If the request would cause the file size to exceed the soft file size limit for the process, the
47463 request shall fail and the implementation shall generate the SIGXFSZ signal for the process.47464 This function shall not modify the file offset for any open file descriptions associated with the
47465 file. Upon successful completion, if the file size is changed, this function shall mark for update
47466 the *st_ctime* and *st_mtime* fields of the file, and the S_ISUID and S_ISGID bits of the file mode
47467 may be cleared.47468 **RETURN VALUE**47469 Upon successful completion, *truncate()* shall return 0. Otherwise, *-1* shall be returned, and *errno*
47470 set to indicate the error.47471 **ERRORS**47472 The *truncate()* function shall fail if:

47473 [EINTR] A signal was caught during execution.

47474 [EINVAL] The *length* argument was less than 0.

47475 [EFBIG] or [EINVAL]

47476 The *length* argument was greater than the maximum file size.

47477 [EIO] An I/O error occurred while reading from or writing to a file system.

47478 [EACCES] A component of the path prefix denies search permission, or write permission
47479 is denied on the file.

47480 [EISDIR] The named file is a directory.

47481 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
47482 argument.

47483 [ENAMETOOLONG]

47484 The length of the *path* argument exceeds {PATH_MAX} or a pathname
47485 component is longer than {NAME_MAX}.47486 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.47487 [ENOTDIR] A component of the path prefix of *path* is not a directory.

47488 [EROFS] The named file resides on a read-only file system.

47489 The *truncate()* function may fail if:

47490 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
47491 resolution of the *path* argument.

47492 [ENAMETOOLONG]
47493 Pathname resolution of a symbolic link produced an intermediate result
47494 whose length exceeds {PATH_MAX}.

47495 **EXAMPLES**

47496 None.

47497 **APPLICATION USAGE**

47498 None.

47499 **RATIONALE**

47500 None.

47501 **FUTURE DIRECTIONS**

47502 None.

47503 **SEE ALSO**

47504 *open()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

47505 **CHANGE HISTORY**

47506 First released in Issue 4, Version 2.

47507 **Issue 5**

47508 Moved from X/OPEN UNIX extension to BASE.

47509 Large File Summit extensions are added.

47510 **Issue 6**

47511 This reference page is split out from the *ftruncate()* reference page.

47512 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

47513 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
47514 [ELOOP] error condition is added.

47515 **NAME**

47516 truncf, trunc — round to truncated integer value

47517 **SYNOPSIS**

47518 #include <math.h>

47519 float truncf(float x);

47520 long double trunc1(long double x);

47521 **DESCRIPTION**47522 Refer to *trunc()*.

47523 NAME

47524 tsearch — search a binary search tree

47525 SYNOPSIS

47526 XSI #include <search.h>

```
47527       void *tsearch(const void *key, void **rootp,  
47528                    int (*compar)(const void *, const void *));
```

47529

47530 DESCRIPTION

47531 Refer to *tdelete()*.

47532 **NAME**

47533 ttyname, ttyname_r — find the pathname of a terminal

47534 **SYNOPSIS**

47535 #include <unistd.h>

47536 char *ttyname(int *fildes*);47537 TSF int ttyname_r(int *fildes*, char **name*, size_t *namesize*);

47538

47539 **DESCRIPTION**

47540 The *ttyname()* function shall return a pointer to a string containing a null-terminated pathname
 47541 of the terminal associated with file descriptor *fildes*. The return value may point to static data
 47542 whose content is overwritten by each call.

47543 The *ttyname()* function need not be reentrant. A function that is not required to be reentrant is
 47544 not required to be thread-safe.

47545 TSF The *ttyname_r()* function shall store the null-terminated pathname of the terminal associated
 47546 with the file descriptor *fildes* in the character array referenced by *name*. The array is *namesize*
 47547 characters long and should have space for the name and the terminating null character. The
 47548 maximum length of the terminal name shall be {TTY_NAME_MAX}.

47549 **RETURN VALUE**

47550 Upon successful completion, *ttyname()* shall return a pointer to a string. Otherwise, a null
 47551 pointer shall be returned and *errno* set to indicate the error.

47552 TSF If successful, the *ttyname_r()* function shall return zero. Otherwise, an error number shall be
 47553 returned to indicate the error.

47554 **ERRORS**47555 The *ttyname()* function may fail if:47556 [EBADF] The *fildes* argument is not a valid file descriptor.47557 [ENOTTY] The *fildes* argument does not refer to a terminal.47558 The *ttyname_r()* function may fail if:47559 TSF [EBADF] The *fildes* argument is not a valid file descriptor.47560 TSF [ENOTTY] The *fildes* argument does not refer to a terminal.

47561 TSF [ERANGE] The value of *namesize* is smaller than the length of the string to be returned
 47562 including the terminating null character.

47563 **EXAMPLES**

47564 None.

47565 **APPLICATION USAGE**

47566 None.

47567 **RATIONALE**

47568 The term “terminal” is used instead of the historical term “terminal device” in order to avoid a
 47569 reference to an undefined term.

47570 The thread-safe version places the terminal name in a user-supplied buffer and returns a non-
 47571 zero value if it fails. The non-thread-safe version may return the name in a static data area that
 47572 may be overwritten by each call.

47573 FUTURE DIRECTIONS

47574 None.

47575 SEE ALSO

47576 The Base Definitions volume of IEEE Std 1003.1-2001, <**unistd.h**>

47577 CHANGE HISTORY

47578 First released in Issue 1. Derived from Issue 1 of the SVID.

47579 Issue 5

47580 The *ttyname_r()* function is included for alignment with the POSIX Threads Extension.

47581 A note indicating that the *ttyname()* function need not be reentrant is added to the
47582 DESCRIPTION.

47583 Issue 6

47584 The *ttyname_r()* function is marked as part of the Thread-Safe Functions option.

47585 The following new requirements on POSIX implementations derive from alignment with the
47586 Single UNIX Specification:

- 47587 • The statement that *errno* is set on error is added.
- 47588 • The [EBADF] and [ENOTTY] optional error conditions are added.

47589 **NAME**

47590 twalk — traverse a binary search tree

47591 **SYNOPSIS**47592 XSI `#include <search.h>`47593 `void twalk(const void *root,`
47594 `void (*action)(const void *, VISIT, int));`

47595

47596 **DESCRIPTION**47597 Refer to *tdelete()*.

47598 **NAME**

47599 daylight, timezone, tzname, tzset — set timezone conversion information

47600 **SYNOPSIS**

47601 #include <time.h>

47602 XSI extern int daylight;

47603 extern long timezone;

47604 CX extern char *tzname[2];

47605 void tzset(void);

47606

47607 **DESCRIPTION**

47608 The *tzset()* function shall use the value of the environment variable *TZ* to set time conversion
 47609 information used by *ctime()*, *localtime()*, *mktime()*, and *strftime()*. If *TZ* is absent from the
 47610 environment, implementation-defined default timezone information shall be used.

47611 The *tzset()* function shall set the external variable *tzname* as follows:

47612 tzname[0] = "std";

47613 tzname[1] = "dst";

47614 where *std* and *dst* are as described in the Base Definitions volume of IEEE Std 1003.1-2001,
 47615 Chapter 8, Environment Variables.

47616 XSI The *tzset()* function also shall set the external variable *daylight* to 0 if Daylight Savings Time
 47617 conversions should never be applied for the timezone in use; otherwise, non-zero. The external
 47618 variable *timezone* shall be set to the difference, in seconds, between Coordinated Universal Time
 47619 (UTC) and local standard time.

47620 **RETURN VALUE**47621 The *tzset()* function shall not return a value.47622 **ERRORS**

47623 No errors are defined.

47624 **EXAMPLES**47625 Example *TZ* variables and their timezone differences are given in the table below:

47626

47627

47628

47629

47630

47631

47632

47633

<i>TZ</i>	<i>timezone</i>
EST5EDT	5*60*60
GMT0	0*60*60
JST-9	-9*60*60
MET-1MEST	-1*60*60
MST7MDT	7*60*60
PST8PDT	8*60*60

47634 **APPLICATION USAGE**

47635 None.

47636 **RATIONALE**

47637 None.

47638 **FUTURE DIRECTIONS**

47639 None.

47640 **SEE ALSO**

47641 *ctime()*, *localtime()*, *mktime()*, *strftime()*, the Base Definitions volume of IEEE Std 1003.1-2001,
47642 Chapter 8, Environment Variables, <**time.h**>

47643 **CHANGE HISTORY**

47644 First released in Issue 1. Derived from Issue 1 of the SVID.

47645 **Issue 6**

47646 The example is corrected.

47647 NAME

47648 `ualarm` — set the interval timer

47649 SYNOPSIS

47650 OB XSI `#include <unistd.h>`

47651 `useconds_t ualarm(useconds_t useconds, useconds_t interval);`

47652

47653 DESCRIPTION

47654 The `ualarm()` function shall cause the SIGALRM signal to be generated for the calling process
 47655 after the number of realtime microseconds specified by the `useconds` argument has elapsed.
 47656 When the `interval` argument is non-zero, repeated timeout notification occurs with a period in
 47657 microseconds specified by the `interval` argument. If the notification signal, SIGALRM, is not
 47658 caught or ignored, the calling process is terminated.

47659 Implementations may place limitations on the granularity of timer values. For each interval
 47660 timer, if the requested timer value requires a finer granularity than the implementation supports,
 47661 the actual timer value shall be rounded up to the next supported value.

47662 Interactions between `ualarm()` and any of the following are unspecified:

47663 `alarm()`
 47664 `nanosleep()`
 47665 `setitimer()`
 47666 `timer_create()`
 47667 `timer_delete()`
 47668 `timer_getoverrun()`
 47669 `timer_gettime()`
 47670 `timer_settime()`
 47671 `sleep()`

47672 RETURN VALUE

47673 The `ualarm()` function shall return the number of microseconds remaining from the previous
 47674 `ualarm()` call. If no timeouts are pending or if `ualarm()` has not previously been called, `ualarm()`
 47675 shall return 0.

47676 ERRORS

47677 No errors are defined.

47678 EXAMPLES

47679 None.

47680 APPLICATION USAGE

47681 Applications are recommended to use `nanosleep()` if the Timers option is supported, or
 47682 `setitimer()`, `timer_create()`, `timer_delete()`, `timer_getoverrun()`, `timer_gettime()`, or `timer_settime()`
 47683 instead of this function.

47684 RATIONALE

47685 None.

47686 FUTURE DIRECTIONS

47687 None.

47688 SEE ALSO

47689 `alarm()`, `nanosleep()`, `setitimer()`, `sleep()`, `timer_create()`, `timer_delete()`, `timer_getoverrun()`, the Base
 47690 Definitions volume of IEEE Std 1003.1-2001, `<unistd.h>`

47691 **CHANGE HISTORY**

47692 First released in Issue 4, Version 2.

47693 **Issue 5**

47694 Moved from X/OPEN UNIX extension to BASE.

47695 **Issue 6**

47696 This function is marked obsolescent.

47697 **NAME**

47698 ulimit — get and set process limits

47699 **SYNOPSIS**

47700 xSI #include <ulimit.h>

47701 long ulimit(int *cmd*, ...);

47702

47703 **DESCRIPTION**

47704 The *ulimit()* function shall control process limits. The process limits that can be controlled by
 47705 this function include the maximum size of a single file that can be written (this is equivalent to
 47706 using *setrlimit()* with *RLIMIT_FSIZE*). The *cmd* values, defined in <ulimit.h>, include:

47707 **UL_GETFSIZE** Return the file size limit (*RLIMIT_FSIZE*) of the process. The limit shall be in
 47708 units of 512-byte blocks and shall be inherited by child processes. Files of any
 47709 size can be read. The return value shall be the integer part of the soft file size
 47710 limit divided by 512. If the result cannot be represented as a **long**, the result is
 47711 unspecified.

47712 **UL_SETFSIZE** Set the file size limit for output operations of the process to the value of the
 47713 second argument, taken as a **long**, multiplied by 512. If the result would
 47714 overflow an **rlim_t**, the actual value set is unspecified. Any process may
 47715 decrease its own limit, but only a process with appropriate privileges may
 47716 increase the limit. The return value shall be the integer part of the new file size
 47717 limit divided by 512.

47718 The *ulimit()* function shall not change the setting of *errno* if successful.

47719 As all return values are permissible in a successful situation, an application wishing to check for
 47720 error situations should set *errno* to 0, then call *ulimit()*, and, if it returns -1 , check to see if *errno* is
 47721 non-zero.

47722 **RETURN VALUE**

47723 Upon successful completion, *ulimit()* shall return the value of the requested limit. Otherwise, -1
 47724 shall be returned and *errno* set to indicate the error.

47725 **ERRORS**

47726 The *ulimit()* function shall fail and the limit shall be unchanged if:

47727 [EINVAL] The *cmd* argument is not valid.

47728 [EPERM] A process not having appropriate privileges attempts to increase its file size
 47729 limit.

47730 **EXAMPLES**

47731 None.

47732 **APPLICATION USAGE**

47733 None.

47734 **RATIONALE**

47735 None.

47736 **FUTURE DIRECTIONS**

47737 None.

47738 **SEE ALSO**

47739 *getrlimit()*, *setrlimit()*, *write()*, the Base Definitions volume of IEEE Std 1003.1-2001, <ulimit.h>

47740 **CHANGE HISTORY**

47741 First released in Issue 1. Derived from Issue 1 of the SVID.

47742 **Issue 5**

47743 In the description of UL_SETFSIZE, the text is corrected to refer to **rlim_t** rather than the
47744 spurious **rlimit_t**.

47745 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

47746 **NAME**

47747 umask — set and get the file mode creation mask

47748 **SYNOPSIS**

47749 #include <sys/stat.h>

47750 mode_t umask(mode_t *cmask*);47751 **DESCRIPTION**

47752 The *umask()* function shall set the process' file mode creation mask to *cmask* and return the
47753 previous value of the mask. Only the file permission bits of *cmask* (see <sys/stat.h>) are used; the
47754 meaning of the other bits is implementation-defined.

47755 The process' file mode creation mask is used during *open()*, *creat()*, *mkdir()*, and *mkfifo()* to turn
47756 off permission bits in the *mode* argument supplied. Bit positions that are set in *cmask* are cleared
47757 in the mode of the created file.

47758 **RETURN VALUE**

47759 The file permission bits in the value returned by *umask()* shall be the previous value of the file
47760 mode creation mask. The state of any other bits in that value is unspecified, except that a
47761 subsequent call to *umask()* with the returned value as *cmask* shall leave the state of the mask the
47762 same as its state before the first call, including any unspecified use of those bits.

47763 **ERRORS**

47764 No errors are defined.

47765 **EXAMPLES**

47766 None.

47767 **APPLICATION USAGE**

47768 None.

47769 **RATIONALE**

47770 Unsigned argument and return types for *umask()* were proposed. The return type and the
47771 argument were both changed to **mode_t**.

47772 Historical implementations have made use of additional bits in *cmask* for their implementation-
47773 defined purposes. The addition of the text that the meaning of other bits of the field is
47774 implementation-defined permits these implementations to conform to this volume of
47775 IEEE Std 1003.1-2001.

47776 **FUTURE DIRECTIONS**

47777 None.

47778 **SEE ALSO**

47779 *creat()*, *mkdir()*, *mkfifo()*, *open()*, the Base Definitions volume of IEEE Std 1003.1-2001,
47780 <sys/stat.h>, <sys/types.h>

47781 **CHANGE HISTORY**

47782 First released in Issue 1. Derived from Issue 1 of the SVID.

47783 **Issue 6**

47784 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

- 47785 The following new requirements on POSIX implementations derive from alignment with the
47786 Single UNIX Specification:
- 47787 • The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was
47788 required for conforming implementations of previous POSIX specifications, it was not
47789 required for UNIX applications.

47790 **NAME**

47791 `uname` — get the name of the current system

47792 **SYNOPSIS**

47793 `#include <sys/utsname.h>`

47794 `int uname(struct utsname *name);`

47795 **DESCRIPTION**

47796 The `uname()` function shall store information identifying the current system in the structure pointed to by `name`.

47798 The `uname()` function uses the **utsname** structure defined in `<sys/utsname.h>`.

47799 The `uname()` function shall return a string naming the current system in the character array `sysname`. Similarly, `nodename` shall contain the name of this node within an implementation-defined communications network. The arrays `release` and `version` shall further identify the operating system. The array `machine` shall contain a name that identifies the hardware that the system is running on.

47804 The format of each member is implementation-defined.

47805 **RETURN VALUE**

47806 Upon successful completion, a non-negative value shall be returned. Otherwise, `-1` shall be returned and `errno` set to indicate the error.

47808 **ERRORS**

47809 No errors are defined.

47810 **EXAMPLES**

47811 None.

47812 **APPLICATION USAGE**

47813 The inclusion of the `nodename` member in this structure does not imply that it is sufficient information for interfacing to communications networks.

47815 **RATIONALE**

47816 The values of the structure members are not constrained to have any relation to the version of this volume of IEEE Std 1003.1-2001 implemented in the operating system. An application should instead depend on `_POSIX_VERSION` and related constants defined in `<unistd.h>`.

47819 This volume of IEEE Std 1003.1-2001 does not define the sizes of the members of the structure and permits them to be of different sizes, although most implementations define them all to be the same size: eight bytes plus one byte for the string terminator. That size for `nodename` is not enough for use with many networks.

47823 The `uname()` function originated in System III, System V, and related implementations, and it does not exist in Version 7 or 4.3 BSD. The values it returns are set at system compile time in those historical implementations.

47826 4.3 BSD has `gethostname()` and `gethostid()`, which return a symbolic name and a numeric value, respectively. There are related `sethostname()` and `sethostid()` functions that are used to set the values the other two functions return. The former functions are included in this specification, the latter are not.

47830 **FUTURE DIRECTIONS**

47831 None.

47832 **SEE ALSO**47833 The Base Definitions volume of IEEE Std 1003.1-2001, <**sys/utsname.h**>47834 **CHANGE HISTORY**

47835 First released in Issue 1. Derived from Issue 1 of the SVID.

47836 **NAME**

47837 ungetc — push byte back into input stream

47838 **SYNOPSIS**

47839 #include <stdio.h>

47840 int ungetc(int *c*, FILE **stream*);47841 **DESCRIPTION**

47842 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 47843 conflict between the requirements described here and the ISO C standard is unintentional. This
 47844 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

47845 The *ungetc()* function shall push the byte specified by *c* (converted to an **unsigned char**) back
 47846 onto the input stream pointed to by *stream*. The pushed-back bytes shall be returned by
 47847 subsequent reads on that stream in the reverse order of their pushing. A successful intervening
 47848 call (with the stream pointed to by *stream*) to a file-positioning function (*fseek()*, *fsetpos()*, or
 47849 *rewind()*) shall discard any pushed-back bytes for the stream. The external storage
 47850 corresponding to the stream shall be unchanged.

47851 One byte of push-back shall be provided. If *ungetc()* is called too many times on the same stream
 47852 without an intervening read or file-positioning operation on that stream, the operation may fail.

47853 If the value of *c* equals that of the macro EOF, the operation shall fail and the input stream shall
 47854 be left unchanged.

47855 A successful call to *ungetc()* shall clear the end-of-file indicator for the stream. The value of the
 47856 file-position indicator for the stream after reading or discarding all pushed-back bytes shall be
 47857 the same as it was before the bytes were pushed back. The file-position indicator is decremented
 47858 by each successful call to *ungetc()*; if its value was 0 before a call, its value is unspecified after
 47859 the call.

47860 **RETURN VALUE**

47861 Upon successful completion, *ungetc()* shall return the byte pushed back after conversion.
 47862 Otherwise, it shall return EOF.

47863 **ERRORS**

47864 No errors are defined.

47865 **EXAMPLES**

47866 None.

47867 **APPLICATION USAGE**

47868 None.

47869 **RATIONALE**

47870 None.

47871 **FUTURE DIRECTIONS**

47872 None.

47873 **SEE ALSO**

47874 *fseek()*, *getc()*, *fsetpos()*, *read()*, *rewind()*, *setbuf()*, the Base Definitions volume of
 47875 IEEE Std 1003.1-2001, <stdio.h>

47876 **CHANGE HISTORY**

47877 First released in Issue 1. Derived from Issue 1 of the SVID.

47878 **NAME**

47879 ungetwc — push wide-character code back into the input stream

47880 **SYNOPSIS**

47881 #include <stdio.h>

47882 #include <wchar.h>

47883 wint_t ungetwc(wint_t wc, FILE *stream);

47884 **DESCRIPTION**

47885 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 47886 conflict between the requirements described here and the ISO C standard is unintentional. This
 47887 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

47888 The *ungetwc()* function shall push the character corresponding to the wide-character code
 47889 specified by *wc* back onto the input stream pointed to by *stream*. The pushed-back characters
 47890 shall be returned by subsequent reads on that stream in the reverse order of their pushing. A
 47891 successful intervening call (with the stream pointed to by *stream*) to a file-positioning function
 47892 (*fseek()*, *fsetpos()*, or *rewind()*) discards any pushed-back characters for the stream. The external
 47893 storage corresponding to the stream is unchanged.

47894 At least one character of push-back shall be provided. If *ungetwc()* is called too many times on
 47895 the same stream without an intervening read or file-positioning operation on that stream, the
 47896 operation may fail.

47897 If the value of *wc* equals that of the macro WEOF, the operation shall fail and the input stream
 47898 shall be left unchanged.

47899 A successful call to *ungetwc()* shall clear the end-of-file indicator for the stream. The value of the
 47900 file-position indicator for the stream after reading or discarding all pushed-back characters shall
 47901 be the same as it was before the characters were pushed back. The file-position indicator is
 47902 decremented (by one or more) by each successful call to *ungetwc()*; if its value was 0 before a
 47903 call, its value is unspecified after the call.

47904 **RETURN VALUE**

47905 Upon successful completion, *ungetwc()* shall return the wide-character code corresponding to
 47906 the pushed-back character. Otherwise, it shall return WEOF.

47907 **ERRORS**47908 The *ungetwc()* function may fail if:

47909 CX [EILSEQ] An invalid character sequence is detected, or a wide-character code does not
 47910 correspond to a valid character.

47911 **EXAMPLES**

47912 None.

47913 **APPLICATION USAGE**

47914 None.

47915 **RATIONALE**

47916 None.

47917 **FUTURE DIRECTIONS**

47918 None.

47919 **SEE ALSO**

47920 *fseek()*, *fsetpos()*, *read()*, *rewind()*, *setbuf()*, the Base Definitions volume of IEEE Std 1003.1-2001,
47921 **<stdio.h>**, **<wchar.h>**

47922 **CHANGE HISTORY**

47923 First released in Issue 4. Derived from the MSE working draft.

47924 **Issue 5**

47925 The Optional Header (OH) marking is removed from **<stdio.h>**.

47926 **Issue 6**

47927 The [EILSEQ] optional error condition is marked CX.

47928 **NAME**47929 **unlink** — remove a directory entry47930 **SYNOPSIS**47931 `#include <unistd.h>`47932 `int unlink(const char *path);`47933 **DESCRIPTION**

47934 The *unlink()* function shall remove a link to a file. If *path* names a symbolic link, *unlink()* shall
 47935 remove the symbolic link named by *path* and shall not affect any file or directory named by the
 47936 contents of the symbolic link. Otherwise, *unlink()* shall remove the link named by the pathname
 47937 pointed to by *path* and shall decrement the link count of the file referenced by the link.

47938 When the file's link count becomes 0 and no process has the file open, the space occupied by the
 47939 file shall be freed and the file shall no longer be accessible. If one or more processes have the file
 47940 open when the last link is removed, the link shall be removed before *unlink()* returns, but the
 47941 removal of the file contents shall be postponed until all references to the file are closed.

47942 The *path* argument shall not name a directory unless the process has appropriate privileges and
 47943 the implementation supports using *unlink()* on directories.

47944 Upon successful completion, *unlink()* shall mark for update the *st_ctime* and *st_mtime* fields of
 47945 the parent directory. Also, if the file's link count is not 0, the *st_ctime* field of the file shall be
 47946 marked for update.

47947 **RETURN VALUE**

47948 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
 47949 indicate the error. If -1 is returned, the named file shall not be changed.

47950 **ERRORS**47951 The *unlink()* function shall fail and shall not unlink the file if:

47952 [EACCES] Search permission is denied for a component of the path prefix, or write
 47953 permission is denied on the directory containing the directory entry to be
 47954 removed.

47955 [EBUSY] The file named by the *path* argument cannot be unlinked because it is being
 47956 used by the system or another process and the implementation considers this
 47957 an error.

47958 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 47959 argument.

47960 [ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX} or a pathname
 47961 component is longer than {NAME_MAX}.
 47962

47963 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

47964 [ENOTDIR] A component of the path prefix is not a directory.

47965 [EPERM] The file named by *path* is a directory, and either the calling process does not
 47966 have appropriate privileges, or the implementation prohibits using *unlink()*
 47967 on directories.

47968 XSI [EPERM] or [EACCES]

47969 The S_ISVTX flag is set on the directory containing the file referred to by the
 47970 *path* argument and the caller is not the file owner, nor is the caller the
 47971 directory owner, nor does the caller have appropriate privileges.

47972 [EROFS] The directory entry to be unlinked is part of a read-only file system.

47973 The *unlink()* function may fail and not unlink the file if:

47974 XSI [EBUSY] The file named by *path* is a named STREAM.

47975 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
47976 resolution of the *path* argument.

47977 [ENAMETOOLONG]
47978 As a result of encountering a symbolic link in resolution of the *path* argument,
47979 the length of the substituted pathname string exceeded {PATH_MAX}.

47980 [ETXTBSY] The entry to be unlinked is the last directory entry to a pure procedure (shared
47981 text) file that is being executed.

47982 EXAMPLES

47983 Removing a Link to a File

47984 The following example shows how to remove a link to a file named **/home/cnd/mod1** by
47985 removing the entry named **/modules/pass1**.

```
47986 #include <unistd.h>
47987
47987 char *path = "/modules/pass1";
47988 int status;
47989 ...
47990 status = unlink(path);
```

47991 Checking for an Error

47992 The following example fragment creates a temporary password lock file named **LOCKFILE**,
47993 which is defined as **/etc/ptmp**, and gets a file descriptor for it. If the file cannot be opened for
47994 writing, *unlink()* is used to remove the link between the file descriptor and **LOCKFILE**.

```
47995 #include <sys/types.h>
47996 #include <stdio.h>
47997 #include <fcntl.h>
47998 #include <errno.h>
47999 #include <unistd.h>
48000 #include <sys/stat.h>
48001
48001 #define LOCKFILE "/etc/ptmp"
48002
48002 int pfd; /* Integer for file descriptor returned by open call. */
48003 FILE *fpfd; /* File pointer for use in putpwent(). */
48004 ...
48005 /* Open password Lock file. If it exists, this is an error. */
48006 if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL, S_IRUSR
48007 | S_IWUSR | S_IRGRP | S_IROTH)) == -1) {
48008     fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
48009     exit(1);
48010 }
48011
48011 /* Lock file created; proceed with fdopen of lock file so that
48012 putpwent() can be used.
48013 */
48014 if ((fpfd = fdopen(pfd, "w")) == NULL) {
```



```

48015         close(pfd);
48016         unlink(LOCKFILE);
48017         exit(1);
48018     }

```

48019 Replacing Files

48020 The following example fragment uses *unlink()* to discard links to files, so that they can be
 48021 replaced with new versions of the files. The first call removes the link to **LOCKFILE** if an error
 48022 occurs. Successive calls remove the links to **SAVEFILE** and **PASSWDFILE** so that new links can
 48023 be created, then removes the link to **LOCKFILE** when it is no longer needed.

```

48024     #include <sys/types.h>
48025     #include <stdio.h>
48026     #include <fcntl.h>
48027     #include <errno.h>
48028     #include <unistd.h>
48029     #include <sys/stat.h>

48030     #define LOCKFILE "/etc/ptmp"
48031     #define PASSWDFILE "/etc/passwd"
48032     #define SAVEFILE "/etc/opasswd"
48033     ...
48034     /* If no change was made, assume error and leave passwd unchanged. */
48035     if (!valid_change) {
48036         fprintf(stderr, "Could not change password for user %s\n", user);
48037         unlink(LOCKFILE);
48038         exit(1);
48039     }

48040     /* Change permissions on new password file. */
48041     chmod(LOCKFILE, S_IRUSR | S_IRGRP | S_IROTH);

48042     /* Remove saved password file. */
48043     unlink(SAVEFILE);

48044     /* Save current password file. */
48045     link(PASSWDFILE, SAVEFILE);

48046     /* Remove current password file. */
48047     unlink(PASSWDFILE);

48048     /* Save new password file as current password file. */
48049     link(LOCKFILE, PASSWDFILE);

48050     /* Remove lock file. */
48051     unlink(LOCKFILE);

48052     exit(0);

```

48053 APPLICATION USAGE

48054 Applications should use *rmdir()* to remove a directory.

48055 RATIONALE

48056 Unlinking a directory is restricted to the superuser in many historical implementations for
 48057 reasons given in *link()* (see also *rename()*).

48058 The meaning of [EBUSY] in historical implementations is “mount point busy”. Since this volume
 48059 of IEEE Std 1003.1-2001 does not cover the system administration concepts of mounting and
 48060 unmounting, the description of the error was changed to “resource busy”. (This meaning is used
 48061 by some device drivers when a second process tries to open an exclusive use device.) The
 48062 wording is also intended to allow implementations to refuse to remove a directory if it is the
 48063 root or current working directory of any process.

48064 FUTURE DIRECTIONS

48065 None.

48066 SEE ALSO

48067 *close()*, *link()*, *remove()*, *rmdir()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**unistd.h**>

48068 CHANGE HISTORY

48069 First released in Issue 1. Derived from Issue 1 of the SVID.

48070 Issue 5

48071 The [EBUSY] error is added to the “may fail” part of the ERRORS section.

48072 Issue 6

48073 The following new requirements on POSIX implementations derive from alignment with the
 48074 Single UNIX Specification:

- 48075 • In the DESCRIPTION, the effect is specified if *path* specifies a symbolic link.
- 48076 • The [ELOOP] mandatory error condition is added.
- 48077 • A second [ENAMETOOLONG] is added as an optional error condition.
- 48078 • The [ETXTBSY] optional error condition is added.

48079 The following changes were made to align with the IEEE P1003.1a draft standard:

- 48080 • The [ELOOP] optional error condition is added.

48081 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

48082 **NAME**

48083 unlockpt — unlock a pseudo-terminal master/slave pair

48084 **SYNOPSIS**

48085 XSI #include <stdlib.h>

48086 int unlockpt(int *fildev*);

48087

48088 **DESCRIPTION**48089 The *unlockpt()* function shall unlock the slave pseudo-terminal device associated with the
48090 master to which *fildev* refers.48091 Conforming applications shall ensure that they call *unlockpt()* before opening the slave side of a
48092 pseudo-terminal device.48093 **RETURN VALUE**48094 Upon successful completion, *unlockpt()* shall return 0. Otherwise, it shall return -1 and set *errno*
48095 to indicate the error.48096 **ERRORS**48097 The *unlockpt()* function may fail if:48098 [EBADF] The *fildev* argument is not a file descriptor open for writing.48099 [EINVAL] The *fildev* argument is not associated with a master pseudo-terminal device.48100 **EXAMPLES**

48101 None.

48102 **APPLICATION USAGE**

48103 None.

48104 **RATIONALE**

48105 None.

48106 **FUTURE DIRECTIONS**

48107 None.

48108 **SEE ALSO**48109 *grantpt()*, *open()*, *ptsname()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>48110 **CHANGE HISTORY**

48111 First released in Issue 4, Version 2.

48112 **Issue 5**

48113 Moved from X/OPEN UNIX extension to BASE.

48114 **Issue 6**

48115 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

48116 **NAME**

48117 unsetenv — remove an environment variable

48118 **SYNOPSIS**48119 **CX** #include <stdlib.h>

48120 int unsetenv(const char *name);

48121

48122 **DESCRIPTION**

48123 The *unsetenv()* function shall remove an environment variable from the environment of the
 48124 calling process. The *name* argument points to a string, which is the name of the variable to be
 48125 removed. The named argument shall not contain an '=' character. If the named variable does
 48126 not exist in the current environment, the environment shall be unchanged and the function is
 48127 considered to have completed successfully.

48128 If the application modifies *environ* or the pointers to which it points, the behavior of *unsetenv()* is
 48129 undefined. The *unsetenv()* function shall update the list of pointers to which *environ* points.

48130 The *unsetenv()* function need not be reentrant. A function that is not required to be reentrant is
 48131 not required to be thread-safe.

48132 **RETURN VALUE**

48133 Upon successful completion, zero shall be returned. Otherwise, -1 shall be returned, *errno* set to
 48134 indicate the error, and the environment shall be unchanged.

48135 **ERRORS**48136 The *unsetenv()* function shall fail if:

48137 [EINVAL] The *name* argument is a null pointer, points to an empty string, or points to a
 48138 string containing an '=' character.

48139 **EXAMPLES**

48140 None.

48141 **APPLICATION USAGE**

48142 None.

48143 **RATIONALE**48144 Refer to the RATIONALE section in *setenv()*.48145 **FUTURE DIRECTIONS**

48146 None.

48147 **SEE ALSO**

48148 *getenv()*, *setenv()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>,
 48149 <sys/types.h>, <unistd.h>

48150 **CHANGE HISTORY**

48151 First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

48152 **NAME**

48153 usleep — suspend execution for an interval

48154 **SYNOPSIS**

48155 OB XSI #include <unistd.h>

48156 int usleep(useconds_t useconds);

48157

48158 **DESCRIPTION**

48159 The *usleep()* function shall cause the calling thread to be suspended from execution until either
 48160 the number of realtime microseconds specified by the argument *useconds* has elapsed or a signal
 48161 is delivered to the calling thread and its action is to invoke a signal-catching function or to
 48162 terminate the process. The suspension time may be longer than requested due to the scheduling
 48163 of other activity by the system.

48164 The *useconds* argument shall be less than one million. If the value of *useconds* is 0, then the call
 48165 has no effect.

48166 If a SIGALRM signal is generated for the calling process during execution of *usleep()* and if the
 48167 SIGALRM signal is being ignored or blocked from delivery, it is unspecified whether *usleep()*
 48168 returns when the SIGALRM signal is scheduled. If the signal is being blocked, it is also
 48169 unspecified whether it remains pending after *usleep()* returns or it is discarded.

48170 If a SIGALRM signal is generated for the calling process during execution of *usleep()*, except as a
 48171 result of a prior call to *alarm()*, and if the SIGALRM signal is not being ignored or blocked from
 48172 delivery, it is unspecified whether that signal has any effect other than causing *usleep()* to return.

48173 If a signal-catching function interrupts *usleep()* and examines or changes either the time a
 48174 SIGALRM is scheduled to be generated, the action associated with the SIGALRM signal, or
 48175 whether the SIGALRM signal is blocked from delivery, the results are unspecified.

48176 If a signal-catching function interrupts *usleep()* and calls *siglongjmp()* or *longjmp()* to restore an
 48177 environment saved prior to the *usleep()* call, the action associated with the SIGALRM signal and
 48178 the time at which a SIGALRM signal is scheduled to be generated are unspecified. It is also
 48179 unspecified whether the SIGALRM signal is blocked, unless the process' signal mask is restored
 48180 as part of the environment.

48181 Implementations may place limitations on the granularity of timer values. For each interval
 48182 timer, if the requested timer value requires a finer granularity than the implementation supports,
 48183 the actual timer value shall be rounded up to the next supported value.

48184 Interactions between *usleep()* and any of the following are unspecified:

48185 *nanosleep()*
 48186 *setitimer()*
 48187 *timer_create()*
 48188 *timer_delete()*
 48189 *timer_getoverrun()*
 48190 *timer_gettime()*
 48191 *timer_settime()*
 48192 *ualarm()*
 48193 *sleep()*

48194 RETURN VALUE

48195 Upon successful completion, *usleep()* shall return 0; otherwise, it shall return -1 and set *errno* to
48196 indicate the error.

48197 ERRORS

48198 The *usleep()* function may fail if:

48199 [EINVAL] The time interval specified one million or more microseconds.

48200 EXAMPLES

48201 None.

48202 APPLICATION USAGE

48203 Applications are recommended to use *nanosleep()* if the Timers option is supported, or
48204 *setitimer()*, *timer_create()*, *timer_delete()*, *timer_getoverrun()*, *timer_gettime()*, or *timer_settime()*
48205 instead of this function.

48206 RATIONALE

48207 None.

48208 FUTURE DIRECTIONS

48209 None.

48210 SEE ALSO

48211 *alarm()*, *getitimer()*, *nanosleep()*, *sigaction()*, *sleep()*, *timer_create()*, *timer_delete()*,
48212 *timer_getoverrun()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**unistd.h**>

48213 CHANGE HISTORY

48214 First released in Issue 4, Version 2.

48215 Issue 5

48216 Moved from X/OPEN UNIX extension to BASE.

48217 The DESCRIPTION is changed to indicate that timers are now thread-based rather than
48218 process-based.

48219 Issue 6

48220 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

48221 This function is marked obsolescent.

48222 **NAME**

48223 utime — set file access and modification times

48224 **SYNOPSIS**

48225 #include <utime.h>

48226 int utime(const char *path, const struct utimbuf *times);

48227 **DESCRIPTION**48228 The *utime()* function shall set the access and modification times of the file named by the *path*
48229 argument.48230 If *times* is a null pointer, the access and modification times of the file shall be set to the current
48231 time. The effective user ID of the process shall match the owner of the file, or the process has
48232 write permission to the file or has appropriate privileges, to use *utime()* in this manner.48233 If *times* is not a null pointer, *times* shall be interpreted as a pointer to a **utimbuf** structure and the
48234 access and modification times shall be set to the values contained in the designated structure.
48235 Only a process with the effective user ID equal to the user ID of the file or a process with
48236 appropriate privileges may use *utime()* this way.48237 The **utimbuf** structure is defined in the <**utime.h**> header. The times in the structure **utimbuf**
48238 are measured in seconds since the Epoch.48239 Upon successful completion, *utime()* shall mark the time of the last file status change, *st_ctime*,
48240 to be updated; see <**sys/stat.h**>.48241 **RETURN VALUE**48242 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* shall
48243 be set to indicate the error, and the file times shall not be affected.48244 **ERRORS**48245 The *utime()* function shall fail if:48246 [EACCES] Search permission is denied by a component of the path prefix; or the *times*
48247 argument is a null pointer and the effective user ID of the process does not
48248 match the owner of the file, the process does not have write permission for the
48249 file, and the process does not have appropriate privileges.48250 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
48251 argument.48252 [ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX} or a pathname
48253 component is longer than {NAME_MAX}.
4825448255 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

48256 [ENOTDIR] A component of the path prefix is not a directory.

48257 [EPERM] The *times* argument is not a null pointer and the calling process' effective user
48258 ID does not match the owner of the file and the calling process does not have
48259 the appropriate privileges.

48260 [EROFS] The file system containing the file is read-only.

48261 The *utime()* function may fail if:48262 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
48263 resolution of the *path* argument.

48264 [ENAMETOOLONG]

48265 As a result of encountering a symbolic link in resolution of the *path* argument,
 48266 the length of the substituted pathname string exceeded {PATH_MAX}.

48267 EXAMPLES

48268 None.

48269 APPLICATION USAGE

48270 None.

48271 RATIONALE

48272 The *actime* structure member must be present so that an application may set it, even though an
 48273 implementation may ignore it and not change the access time on the file. If an application
 48274 intends to leave one of the times of a file unchanged while changing the other, it should use
 48275 *stat()* to retrieve the file's *st_atime* and *st_mtime* parameters, set *actime* and *modtime* in the buffer,
 48276 and change one of them before making the *utime()* call.

48277 FUTURE DIRECTIONS

48278 None.

48279 SEE ALSO

48280 The Base Definitions volume of IEEE Std 1003.1-2001, <sys/stat.h>, <utime.h>

48281 CHANGE HISTORY

48282 First released in Issue 1. Derived from Issue 1 of the SVID.

48283 Issue 6

48284 The following new requirements on POSIX implementations derive from alignment with the
 48285 Single UNIX Specification:

- 48286 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
 48287 required for conforming implementations of previous POSIX specifications, it was not
 48288 required for UNIX applications.
- 48289 • The [ELOOP] mandatory error condition is added.
- 48290 • A second [ENAMETOOLONG] is added as an optional error condition.

48291 The following changes were made to align with the IEEE P1003.1a draft standard:

- 48292 • The [ELOOP] optional error condition is added.

48293 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

48294 **NAME**48295 utimes — set file access and modification times (**LEGACY**)48296 **SYNOPSIS**

48297 xSI #include <sys/time.h>

48298 int utimes(const char *path, const struct timeval times[2]);

48299

48300 **DESCRIPTION**

48301 The *utimes()* function shall set the access and modification times of the file pointed to by the *path*
 48302 argument to the value of the *times* argument. The *utimes()* function allows time specifications
 48303 accurate to the microsecond.

48304 For *utimes()*, the *times* argument is an array of **timeval** structures. The first array member
 48305 represents the date and time of last access, and the second member represents the date and time
 48306 of last modification. The times in the **timeval** structure are measured in seconds and
 48307 microseconds since the Epoch, although rounding toward the nearest second may occur.

48308 If the *times* argument is a null pointer, the access and modification times of the file shall be set to
 48309 the current time. The effective user ID of the process shall match the owner of the file, or has
 48310 write access to the file or appropriate privileges to use this call in this manner. Upon completion,
 48311 *utimes()* shall mark the time of the last file status change, *st_ctime*, for update.

48312 **RETURN VALUE**

48313 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* shall
 48314 be set to indicate the error, and the file times shall not be affected.

48315 **ERRORS**48316 The *utimes()* function shall fail if:

48317 [EACCES] Search permission is denied by a component of the path prefix; or the *times*
 48318 argument is a null pointer and the effective user ID of the process does not
 48319 match the owner of the file and write access is denied.

48320 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 48321 argument.

48322 [ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX} or a pathname
 48323 component is longer than {NAME_MAX}.

48325 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

48326 [ENOTDIR] A component of the path prefix is not a directory.

48327 [EPERM] The *times* argument is not a null pointer and the calling process' effective user
 48328 ID has write access to the file but does not match the owner of the file and the
 48329 calling process does not have the appropriate privileges.

48330 [EROFS] The file system containing the file is read-only.

48331 The *utimes()* function may fail if:

48332 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 48333 resolution of the *path* argument.

48334 [ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result
 48335 whose length exceeds {PATH_MAX}.

48337 EXAMPLES

48338 None.

48339 APPLICATION USAGE

48340 For applications portability, the *utime()* function should be used to set file access and
48341 modification times instead of *utimes()*.

48342 RATIONALE

48343 None.

48344 FUTURE DIRECTIONS

48345 This function may be withdrawn in a future version.

48346 SEE ALSO

48347 *utime()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**sys/time.h**>

48348 CHANGE HISTORY

48349 First released in Issue 4, Version 2.

48350 Issue 5

48351 Moved from X/OPEN UNIX extension to BASE.

48352 Issue 6

48353 This function is marked LEGACY.

48354 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

48355 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
48356 [ELOOP] error condition is added.

48357 **NAME**48358 **va_arg, va_copy, va_end, va_start** — handle variable argument list48359 **SYNOPSIS**48360 `#include <stdarg.h>`48361 `type va_arg(va_list ap, type);`48362 `void va_copy(va_list dest, va_list src);`48363 `void va_end(va_list ap);`48364 `void va_start(va_list ap, argN);`48365 **DESCRIPTION**48366 Refer to the Base Definitions volume of IEEE Std 1003.1-2001, **<stdarg.h>**.

48367 **NAME**

48368 vfork — create a new process; share virtual memory

48369 **SYNOPSIS**

48370 OB XSI #include <unistd.h>

48371 pid_t vfork(void);

48372

48373 **DESCRIPTION**

48374 The *vfork()* function shall be equivalent to *fork()*, except that the behavior is undefined if the
 48375 process created by *vfork()* either modifies any data other than a variable of type **pid_t** used to
 48376 store the return value from *vfork()*, or returns from the function in which *vfork()* was called, or
 48377 calls any other function before successfully calling *_exit()* or one of the *exec* family of functions.

48378 **RETURN VALUE**

48379 Upon successful completion, *vfork()* shall return 0 to the child process and return the process ID
 48380 of the child process to the parent process. Otherwise, -1 shall be returned to the parent, no child
 48381 process shall be created, and *errno* shall be set to indicate the error.

48382 **ERRORS**48383 The *vfork()* function shall fail if:

48384 [EAGAIN] The system-wide limit on the total number of processes under execution
 48385 would be exceeded, or the system-imposed limit on the total number of
 48386 processes under execution by a single user would be exceeded.

48387 [ENOMEM] There is insufficient swap space for the new process.

48388 **EXAMPLES**

48389 None.

48390 **APPLICATION USAGE**

48391 Conforming applications are recommended not to depend on *vfork()*, but to use *fork()* instead.
 48392 The *vfork()* function may be withdrawn in a future version.

48393 On some implementations, *vfork()* is equivalent to *fork()*.

48394 The *vfork()* function differs from *fork()* only in that the child process can share code and data
 48395 with the calling process (parent process). This speeds cloning activity significantly at a risk to
 48396 the integrity of the parent process if *vfork()* is misused.

48397 The use of *vfork()* for any purpose except as a prelude to an immediate call to a function from
 48398 the *exec* family, or to *_exit()*, is not advised.

48399 The *vfork()* function can be used to create new processes without fully copying the address
 48400 space of the old process. If a forked process is simply going to call *exec*, the data space copied
 48401 from the parent to the child by *fork()* is not used. This is particularly inefficient in a paged
 48402 environment, making *vfork()* particularly useful. Depending upon the size of the parent's data
 48403 space, *vfork()* can give a significant performance improvement over *fork()*.

48404 The *vfork()* function can normally be used just like *fork()*. It does not work, however, to return
 48405 while running in the child's context from the caller of *vfork()* since the eventual return from
 48406 *vfork()* would then return to a no longer existent stack frame. Care should be taken, also, to call
 48407 *_exit()* rather than *exit()* if *exec* cannot be used, since *exit()* flushes and closes standard I/O
 48408 channels, thereby damaging the parent process' standard I/O data structures. (Even with *fork()*,
 48409 it is wrong to call *exit()*, since buffered data would then be flushed twice.)

48410 If signal handlers are invoked in the child process after *vfork()*, they must follow the same rules
 48411 as other code in the child process.

48412 **RATIONALE**

48413 None.

48414 **FUTURE DIRECTIONS**

48415 This function may be withdrawn in a future version.

48416 **SEE ALSO**48417 *exec*, *exit()*, *fork()*, *wait()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**unistd.h**>48418 **CHANGE HISTORY**

48419 First released in Issue 4, Version 2.

48420 **Issue 5**

48421 Moved from X/OPEN UNIX extension to BASE.

48422 **Issue 6**

48423 This function is marked obsolescent.

48424 **NAME**

48425 vfprintf, vprintf, vsnprintf, vsprintf — format output of a stdarg argument list

48426 **SYNOPSIS**

48427 #include <stdarg.h>

48428 #include <stdio.h>

48429 int vfprintf(FILE *restrict *stream*, const char *restrict *format*,
48430 va_list *ap*);48431 int vprintf(const char *restrict *format*, va_list *ap*);48432 int vsnprintf(char *restrict *s*, size_t *n*, const char *restrict *format*,
48433 va_list *ap*);48434 int vsprintf(char *restrict *s*, const char *restrict *format*, va_list *ap*);48435 **DESCRIPTION**48436 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
48437 conflict between the requirements described here and the ISO C standard is unintentional. This
48438 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.48439 The *vprintf()*, *vfprintf()*, *vsnprintf()*, and *vsprintf()* functions shall be equivalent to *printf()*,
48440 *fprintf()*, *snprintf()*, and *sprintf()* respectively, except that instead of being called with a variable
48441 number of arguments, they are called with an argument list as defined by <stdarg.h>.48442 These functions shall not invoke the *va_end* macro. As these functions invoke the *va_arg* macro,
48443 the value of *ap* after the return is unspecified.48444 **RETURN VALUE**48445 Refer to *fprintf()*.48446 **ERRORS**48447 Refer to *fprintf()*.48448 **EXAMPLES**

48449 None.

48450 **APPLICATION USAGE**48451 Applications using these functions should call *va_end(ap)* afterwards to clean up.48452 **RATIONALE**

48453 None.

48454 **FUTURE DIRECTIONS**

48455 None.

48456 **SEE ALSO**48457 *fprintf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdarg.h>, <stdio.h>48458 **CHANGE HISTORY**

48459 First released in Issue 1. Derived from Issue 1 of the SVID.

48460 **Issue 5**48461 The *vsnprintf()* function is added.48462 **Issue 6**48463 The *vfprintf()*, *vprintf()*, *vsnprintf()*, and *vsprintf()* functions are updated for alignment with the
48464 ISO/IEC 9899:1999 standard.

48465 **NAME**

48466 vfscanf, vscanf, vsscanf — format input of a stdarg argument list

48467 **SYNOPSIS**

48468 #include <stdarg.h>

48469 #include <stdio.h>

48470 int vfscanf(FILE *restrict stream, const char *restrict format,
48471 va_list arg);

48472 int vscanf(const char *restrict format, va_list arg);

48473 int vsscanf(const char *restrict s, const char *restrict format,

48474 va_list arg);

48475 **DESCRIPTION**48476 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
48477 conflict between the requirements described here and the ISO C standard is unintentional. This
48478 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.48479 The *vscanf()*, *vfscanf()*, and *vsscanf()* functions shall be equivalent to the *scanf()*, *fscanf()*, and
48480 *sscanf()* functions, respectively, except that instead of being called with a variable number of
48481 arguments, they are called with an argument list as defined in the <stdarg.h> header. These
48482 functions shall not invoke the *va_end* macro. As these functions invoke the *va_arg* macro, the
48483 value of *ap* after the return is unspecified.48484 **RETURN VALUE**48485 Refer to *fscanf()*.48486 **ERRORS**48487 Refer to *fscanf()*.48488 **EXAMPLES**

48489 None.

48490 **APPLICATION USAGE**48491 Applications using these functions should call *va_end(ap)* afterwards to clean up.48492 **RATIONALE**

48493 None.

48494 **FUTURE DIRECTIONS**

48495 None.

48496 **SEE ALSO**48497 *fscanf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdarg.h>, <stdio.h>48498 **CHANGE HISTORY**

48499 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

48500 **NAME**

48501 vfwprintf, vswprintf, vwprintf — wide-character formatted output of a stdarg argument list

48502 **SYNOPSIS**

48503 #include <stdarg.h>

48504 #include <stdio.h>

48505 #include <wchar.h>

48506 int vfwprintf(FILE *restrict stream, const wchar_t *restrict format,
48507 va_list arg);

48508 int vswprintf(wchar_t *restrict ws, size_t n,

48509 const wchar_t *restrict format, va_list arg);

48510 int vwprintf(const wchar_t *restrict format, va_list arg);

48511 **DESCRIPTION**48512 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
48513 conflict between the requirements described here and the ISO C standard is unintentional. This
48514 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.48515 The *vfwprintf()*, *vswprintf()*, and *vwprintf()* functions shall be equivalent to *fwprintf()*, *swprintf()*,
48516 and *wprintf()* respectively, except that instead of being called with a variable number of
48517 arguments, they are called with an argument list as defined by <stdarg.h>.48518 These functions shall not invoke the *va_end* macro. However, as these functions do invoke the
48519 *va_arg* macro, the value of *ap* after the return is unspecified.48520 **RETURN VALUE**48521 Refer to *fwprintf()*.48522 **ERRORS**48523 Refer to *fwprintf()*.48524 **EXAMPLES**

48525 None.

48526 **APPLICATION USAGE**48527 Applications using these functions should call *va_end(ap)* afterwards to clean up.48528 **RATIONALE**

48529 None.

48530 **FUTURE DIRECTIONS**

48531 None.

48532 **SEE ALSO**48533 *fwprintf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdarg.h>, <stdio.h>,
48534 <wchar.h>48535 **CHANGE HISTORY**48536 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
48537 (E).48538 **Issue 6**48539 The *vfwprintf()*, *vswprintf()*, and *vwprintf()* prototypes are updated for alignment with the
48540 ISO/IEC 9899:1999 standard. ()

48541 **NAME**

48542 vfwscanf, vswscanf, vwscanf — wide-character formatted input of a stdarg argument list

48543 **SYNOPSIS**

48544 #include <stdarg.h>

48545 #include <stdio.h>

48546 #include <wchar.h>

48547 int vfwscanf(FILE *restrict stream, const wchar_t *restrict format,
48548 va_list arg);48549 int vswscanf(const wchar_t *restrict ws, const wchar_t *restrict format,
48550 va_list arg);

48551 int vwscanf(const wchar_t *restrict format, va_list arg);

48552 **DESCRIPTION**48553 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
48554 conflict between the requirements described here and the ISO C standard is unintentional. This
48555 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.48556 The *vfwscanf()*, *vswscanf()*, and *vwscanf()* functions shall be equivalent to the *fwscanf()*,
48557 *swscanf()*, and *wscanf()* functions, respectively, except that instead of being called with a
48558 variable number of arguments, they are called with an argument list as defined in the <stdarg.h>
48559 header. These functions shall not invoke the *va_end* macro. As these functions invoke the *va_arg*
48560 macro, the value of *ap* after the return is unspecified.48561 **RETURN VALUE**48562 Refer to *fwscanf()*.48563 **ERRORS**48564 Refer to *fwscanf()*.48565 **EXAMPLES**

48566 None.

48567 **APPLICATION USAGE**48568 Applications using these functions should call *va_end(ap)* afterwards to clean up.48569 **RATIONALE**

48570 None.

48571 **FUTURE DIRECTIONS**

48572 None.

48573 **SEE ALSO**48574 *fwscanf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdarg.h>, <stdio.h>,
48575 <wchar.h>48576 **CHANGE HISTORY**

48577 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

48578 **NAME**

48579 vprintf — format the output of a stdarg argument list

48580 **SYNOPSIS**

48581 #include <stdarg.h>

48582 #include <stdio.h>

48583 int vprintf(const char *restrict *format*, va_list *ap*);48584 **DESCRIPTION**48585 Refer to *vfprintf()*.

48586 **NAME**

48587 vscanf — format input of a stdarg argument list

48588 **SYNOPSIS**

48589 #include <stdarg.h>

48590 #include <stdio.h>

48591 int vscanf(const char *restrict *format*, va_list *arg*);48592 **DESCRIPTION**48593 Refer to *vfscanf()*.

48594 **NAME**

48595 vsprintf, vsprintf — format output of a stdarg argument list

48596 **SYNOPSIS**

48597 #include <stdarg.h>

48598 #include <stdio.h>

48599 int vsnprintf(char *restrict *s*, size_t *n*,48600 const char *restrict *format*, va_list *ap*);48601 int vsprintf(char *restrict *s*, const char *restrict *format*,48602 va_list *ap*);48603 **DESCRIPTION**48604 Refer to *vfprintf()*.

48605 **NAME**

48606 vsscanf — format input of a stdarg argument list

48607 **SYNOPSIS**

48608 #include <stdarg.h>

48609 #include <stdio.h>

48610 int vsscanf(const char *restrict *s*, const char *restrict *format*,48611 va_list *arg*);48612 **DESCRIPTION**48613 Refer to *vfscanf()*.

48614 **NAME**

48615 vswprintf — wide-character formatted output of a stdarg argument list

48616 **SYNOPSIS**

48617 #include <stdarg.h>

48618 #include <stdio.h>

48619 #include <wchar.h>

```
48620       int vswprintf(wchar_t *restrict ws, size_t n,  
48621                    const wchar_t *restrict format, va_list arg);
```

48622 **DESCRIPTION**48623 Refer to *vfwprintf()*.

48624 **NAME**

48625 vswscanf — wide-character formatted input of a stdarg argument list

48626 **SYNOPSIS**

48627 #include <stdarg.h>

48628 #include <stdio.h>

48629 #include <wchar.h>

48630 int vswscanf(const wchar_t *restrict *ws*, const wchar_t *restrict *format*,48631 va_list *arg*);48632 **DESCRIPTION**48633 Refer to *vfwscanf()*.

48634 **NAME**

48635 vwprintf — wide-character formatted output of a stdarg argument list

48636 **SYNOPSIS**

48637 #include <stdarg.h>

48638 #include <stdio.h>

48639 #include <wchar.h>

48640 int vwprintf(const wchar_t *restrict *format*, va_list *arg*);48641 **DESCRIPTION**48642 Refer to *vfwprintf()*.

48643 **NAME**

48644 vwscanf — wide-character formatted input of a stdarg argument list

48645 **SYNOPSIS**

48646 #include <stdarg.h>

48647 #include <stdio.h>

48648 #include <wchar.h>

48649 int vwscanf(const wchar_t *restrict *format*, va_list *arg*);48650 **DESCRIPTION**48651 Refer to *vfwscanf()*.

48652 NAME

48653 wait, waitpid — wait for a child process to stop or terminate

48654 SYNOPSIS

48655 #include <sys/wait.h>

48656 pid_t wait(int *stat_loc);

48657 pid_t waitpid(pid_t pid, int *stat_loc, int options);

48658 DESCRIPTION

48659 The *wait()* and *waitpid()* functions shall obtain status information pertaining to one of the
 48660 caller's child processes. Various options permit status information to be obtained for child
 48661 processes that have terminated or stopped. If status information is available for two or more
 48662 child processes, the order in which their status is reported is unspecified.

48663 The *wait()* function shall suspend execution of the calling thread until status information for one
 48664 of the terminated child processes of the calling process is available, or until delivery of a signal
 48665 whose action is either to execute a signal-catching function or to terminate the process. If more
 48666 than one thread is suspended in *wait()* or *waitpid()* awaiting termination of the same process,
 48667 exactly one thread shall return the process status at the time of the target process termination. If
 48668 status information is available prior to the call to *wait()*, return shall be immediate.

48669 The *waitpid()* function shall be equivalent to *wait()* if the *pid* argument is (**pid_t**)−1 and the
 48670 *options* argument is 0. Otherwise, its behavior shall be modified by the values of the *pid* and
 48671 *options* arguments.

48672 The *pid* argument specifies a set of child processes for which *status* is requested. The *waitpid()*
 48673 function shall only return the status of a child process from this set:

- 48674 • If *pid* is equal to (**pid_t**)−1, *status* is requested for any child process. In this respect, *waitpid()*
 48675 is then equivalent to *wait()*.
- 48676 • If *pid* is greater than 0, it specifies the process ID of a single child process for which *status* is
 48677 requested.
- 48678 • If *pid* is 0, *status* is requested for any child process whose process group ID is equal to that of
 48679 the calling process.
- 48680 • If *pid* is less than (**pid_t**)−1, *status* is requested for any child process whose process group ID
 48681 is equal to the absolute value of *pid*.

48682 The *options* argument is constructed from the bitwise-inclusive OR of zero or more of the
 48683 following flags, defined in the <sys/wait.h> header:

48684 XSI WCONTINUED The *waitpid()* function shall report the status of any continued child process
 48685 specified by *pid* whose status has not been reported since it continued from a
 48686 job control stop.

48687 WNOHANG The *waitpid()* function shall not suspend execution of the calling thread if
 48688 *status* is not immediately available for one of the child processes specified by
 48689 *pid*.

48690 WUNTRACED The status of any child processes specified by *pid* that are stopped, and whose
 48691 status has not yet been reported since they stopped, shall also be reported to
 48692 the requesting process.

48693 XSI If the calling process has SA_NOCLDWAIT set or has SIGCHLD set to SIG_IGN, and the
 48694 process has no unwaited-for children that were transformed into zombie processes, the calling
 48695 thread shall block until all of the children of the process containing the calling thread terminate,
 48696 and *wait()* and *waitpid()* shall fail and set *errno* to [ECHILD].

If *wait()* or *waitpid()* return because the status of a child process is available, these functions shall return a value equal to the process ID of the child process. In this case, if the value of the argument *stat_loc* is not a null pointer, information shall be stored in the location pointed to by *stat_loc*. The value stored at the location pointed to by *stat_loc* shall be 0 if and only if the status returned is from a terminated child process that terminated by one of the following means:

1. The process returned 0 from *main()*.
2. The process called *_exit()* or *exit()* with a *status* argument of 0.
3. The process was terminated because the last thread in the process terminated.

Regardless of its value, this information may be interpreted using the following macros, which are defined in `<sys/wait.h>` and evaluate to integral expressions; the *stat_val* argument is the integer value pointed to by *stat_loc*.

WIFEXITED(*stat_val*)

Evaluates to a non-zero value if *status* was returned for a child process that terminated normally.

WEXITSTATUS(*stat_val*)

If the value of **WIFEXITED(*stat_val*)** is non-zero, this macro evaluates to the low-order 8 bits of the *status* argument that the child process passed to *_exit()* or *exit()*, or the value the child process returned from *main()*.

WIFSIGNALED(*stat_val*)

Evaluates to a non-zero value if *status* was returned for a child process that terminated due to the receipt of a signal that was not caught (see `<signal.h>`).

WTERMSIG(*stat_val*)

If the value of **WIFSIGNALED(*stat_val*)** is non-zero, this macro evaluates to the number of the signal that caused the termination of the child process.

WIFSTOPPED(*stat_val*)

Evaluates to a non-zero value if *status* was returned for a child process that is currently stopped.

WSTOPSIG(*stat_val*)

If the value of **WIFSTOPPED(*stat_val*)** is non-zero, this macro evaluates to the number of the signal that caused the child process to stop.

WIFCONTINUED(*stat_val*)

Evaluates to a non-zero value if *status* was returned for a child process that has continued from a job control stop.

It is unspecified whether the *status* value returned by calls to *wait()* or *waitpid()* for processes created by *posix_spawn()* or *posix_spawnnp()* can indicate a **WIFSTOPPED(*stat_val*)** before subsequent calls to *wait()* or *waitpid()* indicate **WIFEXITED(*stat_val*)** as the result of an error detected before the new process image starts executing.

It is unspecified whether the *status* value returned by calls to *wait()* or *waitpid()* for processes created by *posix_spawn()* or *posix_spawnnp()* can indicate a **WIFSIGNALED(*stat_val*)** if a signal is sent to the parent's process group after *posix_spawn()* or *posix_spawnnp()* is called.

If the information pointed to by *stat_loc* was stored by a call to *waitpid()* that specified the **WUNTRACED** flag and did not specify the **WCONTINUED** flag, exactly one of the macros **WIFEXITED(**stat_loc*)**, **WIFSIGNALED(**stat_loc*)**, and **WIFSTOPPED(**stat_loc*)** shall evaluate to a non-zero value.

48741 If the information pointed to by *stat_loc* was stored by a call to *waitpid()* that specified the
 48742 XSI WUNTRACED and WCONTINUED flags, exactly one of the macros WIFEXITED(**stat_loc*),
 48743 XSI WIFSIGNALED(**stat_loc*), WIFSTOPPED(**stat_loc*), and WIFCONTINUED(**stat_loc*) shall
 48744 evaluate to a non-zero value.

48745 If the information pointed to by *stat_loc* was stored by a call to *waitpid()* that did not specify the
 48746 XSI WUNTRACED or WCONTINUED flags, or by a call to the *wait()* function, exactly one of the
 48747 macros WIFEXITED(**stat_loc*) and WIFSIGNALED(**stat_loc*) shall evaluate to a non-zero value.

48748 If the information pointed to by *stat_loc* was stored by a call to *waitpid()* that did not specify the
 48749 XSI WUNTRACED flag and specified the WCONTINUED flag, or by a call to the *wait()* function,
 48750 XSI exactly one of the macros WIFEXITED(**stat_loc*), WIFSIGNALED(**stat_loc*), and
 48751 WIFCONTINUED(**stat_loc*) shall evaluate to a non-zero value.

48752 If _POSIX_REALTIME_SIGNALS is defined, and the implementation queues the SIGCHLD
 48753 signal, then if *wait()* or *waitpid()* returns because the status of a child process is available, any
 48754 pending SIGCHLD signal associated with the process ID of the child process shall be discarded.
 48755 Any other pending SIGCHLD signals shall remain pending.

48756 Otherwise, if SIGCHLD is blocked, if *wait()* or *waitpid()* return because the status of a child
 48757 process is available, any pending SIGCHLD signal shall be cleared unless the status of another
 48758 child process is available.

48759 For all other conditions, it is unspecified whether child *status* will be available when a SIGCHLD
 48760 signal is delivered.

48761 There may be additional implementation-defined circumstances under which *wait()* or *waitpid()*
 48762 report *status*. This shall not occur unless the calling process or one of its child processes explicitly
 48763 makes use of a non-standard extension. In these cases the interpretation of the reported *status* is
 48764 implementation-defined.

48765 XSI If a parent process terminates without waiting for all of its child processes to terminate, the
 48766 remaining child processes shall be assigned a new parent process ID corresponding to an
 48767 implementation-defined system process.

48768 RETURN VALUE

48769 If *wait()* or *waitpid()* returns because the status of a child process is available, these functions
 48770 shall return a value equal to the process ID of the child process for which *status* is reported. If
 48771 *wait()* or *waitpid()* returns due to the delivery of a signal to the calling process, -1 shall be
 48772 returned and *errno* set to [EINTR]. If *waitpid()* was invoked with WNOHANG set in *options*, it
 48773 has at least one child process specified by *pid* for which *status* is not available, and *status* is not
 48774 available for any process specified by *pid*, 0 is returned. Otherwise, (pid_t)-1 shall be returned,
 48775 and *errno* set to indicate the error.

48776 ERRORS

48777 The *wait()* function shall fail if:

48778 [ECHILD] The calling process has no existing unwaited-for child processes.

48779 [EINTR] The function was interrupted by a signal. The value of the location pointed to
 48780 by *stat_loc* is undefined.

48781 The *waitpid()* function shall fail if:

48782 [ECHILD] The process specified by *pid* does not exist or is not a child of the calling
 48783 process, or the process group specified by *pid* does not exist or does not have
 48784 any member process that is a child of the calling process.

48785 [EINTR] The function was interrupted by a signal. The value of the location pointed to
 48786 by *stat_loc* is undefined.

48787 [EINVAL] The *options* argument is not valid.

48788 EXAMPLES

48789 None.

48790 APPLICATION USAGE

48791 None.

48792 RATIONALE

48793 A call to the *wait()* or *waitpid()* function only returns *status* on an immediate child process of the
 48794 calling process; that is, a child that was produced by a single *fork()* call (perhaps followed by an
 48795 *exec* or other function calls) from the parent. If a child produces grandchildren by further use of
 48796 *fork()*, none of those grandchildren nor any of their descendants affect the behavior of a *wait()*
 48797 from the original parent process. Nothing in this volume of IEEE Std 1003.1-2001 prevents an
 48798 implementation from providing extensions that permit a process to get *status* from a grandchild
 48799 or any other process, but a process that does not use such extensions must be guaranteed to see
 48800 *status* from only its direct children.

48801 The *waitpid()* function is provided for three reasons:

- 48802 1. To support job control
- 48803 2. To permit a non-blocking version of the *wait()* function
- 48804 3. To permit a library routine, such as *system()* or *pclose()*, to wait for its children without
 48805 interfering with other terminated children for which the process has not waited

48806 The first two of these facilities are based on the *wait3()* function provided by 4.3 BSD. The
 48807 function uses the *options* argument, which is equivalent to an argument to *wait3()*. The
 48808 WUNTRACED flag is used only in conjunction with job control on systems supporting job
 48809 control. Its name comes from 4.3 BSD and refers to the fact that there are two types of stopped
 48810 processes in that implementation: processes being traced via the *ptrace()* debugging facility and
 48811 (untraced) processes stopped by job control signals. Since *ptrace()* is not part of this volume of
 48812 IEEE Std 1003.1-2001, only the second type is relevant. The name WUNTRACED was retained
 48813 because its usage is the same, even though the name is not intuitively meaningful in this context.

48814 The third reason for the *waitpid()* function is to permit independent sections of a process to
 48815 spawn and wait for children without interfering with each other. For example, the following
 48816 problem occurs in developing a portable shell, or command interpreter:

```
48817 stream = popen("/bin/true");
48818 (void) system("sleep 100");
48819 (void) pclose(stream);
```

48820 On all historical implementations, the final *pclose()* fails to reap the *wait()* *status* of the *popen()*.

48821 The status values are retrieved by macros, rather than given as specific bit encodings as they are
 48822 in most historical implementations (and thus expected by existing programs). This was
 48823 necessary to eliminate a limitation on the number of signals an implementation can support that
 48824 was inherent in the traditional encodings. This volume of IEEE Std 1003.1-2001 does require that
 48825 a *status* value of zero corresponds to a process calling *_exit(0)*, as this is the most common
 48826 encoding expected by existing programs. Some of the macro names were adopted from 4.3 BSD.

48827 These macros syntactically operate on an arbitrary integer value. The behavior is undefined
 48828 unless that value is one stored by a successful call to *wait()* or *waitpid()* in the location pointed
 48829 to by the *stat_loc* argument. An early proposal attempted to make this clearer by specifying each

argument as **stat_loc* rather than *stat_val*. However, that did not follow the conventions of other specifications in this volume of IEEE Std 1003.1-2001 or traditional usage. It also could have implied that the argument to the macro must literally be **stat_loc*; in fact, that value can be stored or passed as an argument to other functions before being interpreted by these macros.

The extension that affects *wait()* and *waitpid()* and is common in historical implementations is the *ptrace()* function. It is called by a child process and causes that child to stop and return a *status* that appears identical to the *status* indicated by WIFSTOPPED. The *status* of *ptrace()* children is traditionally returned regardless of the WUNTRACED flag (or by the *wait()* function). Most applications do not need to concern themselves with such extensions because they have control over what extensions they or their children use. However, applications, such as command interpreters, that invoke arbitrary processes may see this behavior when those arbitrary processes misuse such extensions.

Implementations that support **core** file creation or other implementation-defined actions on termination of some processes traditionally provide a bit in the *status* returned by *wait()* to indicate that such actions have occurred.

Allowing the *wait()* family of functions to discard a pending SIGCHLD signal that is associated with a successfully waited-for child process puts them into the *sigwait()* and *sigwaitinfo()* category with respect to SIGCHLD.

This definition allows implementations to treat a pending SIGCHLD signal as accepted by the process in *wait()*, with the same meaning of “accepted” as when that word is applied to the *sigwait()* family of functions.

Allowing the *wait()* family of functions to behave this way permits an implementation to be able to deal precisely with SIGCHLD signals.

In particular, an implementation that does accept (discard) the SIGCHLD signal can make the following guarantees regardless of the queuing depth of signals in general (the list of waitable children can hold the SIGCHLD queue):

1. If a SIGCHLD signal handler is established via *sigaction()* without the SA_RESETHAND flag, SIGCHLD signals can be accurately counted; that is, exactly one SIGCHLD signal will be delivered to or accepted by the process for every child process that terminates.
2. A single *wait()* issued from a SIGCHLD signal handler can be guaranteed to return immediately with status information for a child process.
3. When SA_SIGINFO is requested, the SIGCHLD signal handler can be guaranteed to receive a non-NULL pointer to a **siginfo_t** structure that describes a child process for which a wait via *waitpid()* or *waitid()* will not block or fail.
4. The *system()* function will not cause a process' SIGCHLD handler to be called as a result of the *fork()/exec* executed within *system()* because *system()* will accept the SIGCHLD signal when it performs a *waitpid()* for its child process. This is a desirable behavior of *system()* so that it can be used in a library without causing side effects to the application linked with the library.

An implementation that does not permit the *wait()* family of functions to accept (discard) a pending SIGCHLD signal associated with a successfully waited-for child, cannot make the guarantees described above for the following reasons:

Guarantee #1

Although it might be assumed that reliable queuing of all SIGCHLD signals generated by the system can make this guarantee, the counter-example is the case of a process that blocks SIGCHLD and performs an indefinite loop of *fork()/wait()* operations. If the

48876 implementation supports queued signals, then eventually the system will run out of
 48877 memory for the queue. The guarantee cannot be made because there must be some limit to
 48878 the depth of queuing.

48879 Guarantees #2 and #3

48880 These cannot be guaranteed unless the *wait()* family of functions accepts the SIGCHLD
 48881 signal. Otherwise, a *fork()/wait()* executed while SIGCHLD is blocked (as in the *system()*
 48882 function) will result in an invocation of the handler when SIGCHLD is unblocked, after the
 48883 process has disappeared.

48884 Guarantee #4

48885 Although possible to make this guarantee, *system()* would have to set the SIGCHLD
 48886 handler to SIG_DFL so that the SIGCHLD signal generated by its *fork()* would be discarded
 48887 (the SIGCHLD default action is to be ignored), then restore it to its previous setting. This
 48888 would have the undesirable side effect of discarding all SIGCHLD signals pending to the
 48889 process.

48890 FUTURE DIRECTIONS

48891 None.

48892 SEE ALSO

48893 *exec*, *exit()*, *fork()*, *waitid()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**signal.h**>,
 48894 <**sys/wait.h**>

48895 CHANGE HISTORY

48896 First released in Issue 1. Derived from Issue 1 of the SVID.

48897 Issue 5

48898 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

48899 Issue 6

48900 The following new requirements on POSIX implementations derive from alignment with the
 48901 Single UNIX Specification:

- 48902 • The requirement to include <**sys/types.h**> has been removed. Although <**sys/types.h**> was
 48903 required for conforming implementations of previous POSIX specifications, it was not
 48904 required for UNIX applications.

48905 The following changes were made to align with the IEEE P1003.1a draft standard:

- 48906 • The processing of the SIGCHLD signal and the [ECHILD] error is clarified.

48907 The semantics of WIFSTOPPED(*stat_val*), WIFEXITED(*stat_val*), and WIFSIGNALED(*stat_val*)
 48908 are defined with respect to *posix_spawn()* or *posix_spawnnp()* for alignment with
 48909 IEEE Std 1003.1d-1999.

48910 The DESCRIPTION is updated for alignment with the ISO/IEC 9899:1999 standard.

48911 NAME

48912 waitid — wait for a child process to change state

48913 SYNOPSIS

48914 XSI `#include <sys/wait.h>`48915 `int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);`

48916

48917 DESCRIPTION

48918 The *waitid()* function shall suspend the calling thread until one child of the process containing
 48919 the calling thread changes state. It records the current state of a child in the structure pointed to
 48920 by *infop*. If a child process changed state prior to the call to *waitid()*, *waitid()* shall return
 48921 immediately. If more than one thread is suspended in *wait()* or *waitpid()* waiting for termination
 48922 of the same process, exactly one thread shall return the process status at the time of the target
 48923 process termination.

48924 The *idtype* and *id* arguments are used to specify which children *waitid()* waits for.

48925 If *idtype* is P_PID, *waitid()* shall wait for the child with a process ID equal to (**pid_t**)*id*.

48926 If *idtype* is P_PGID, *waitid()* shall wait for any child with a process group ID equal to (**pid_t**)*id*.

48927 If *idtype* is P_ALL, *waitid()* shall wait for any children and *id* is ignored.

48928 The *options* argument is used to specify which state changes *waitid()* shall wait for. It is formed
 48929 by OR'ing together one or more of the following flags:

48930 WEXITED Wait for processes that have exited.

48931 WSTOPPED Status shall be returned for any child that has stopped upon receipt of a signal.

48932 WCONTINUED Status shall be returned for any child that was stopped and has been
 48933 continued.

48934 WNOHANG Return immediately if there are no children to wait for.

48935 WNOWAIT Keep the process whose status is returned in *infop* in a waitable state. This
 48936 shall not affect the state of the process; the process may be waited for again
 48937 after this call completes.

48938 The application shall ensure that the *infop* argument points to a **siginfo_t** structure. If *waitid()*
 48939 returns because a child process was found that satisfied the conditions indicated by the
 48940 arguments *idtype* and *options*, then the structure pointed to by *infop* shall be filled in by the
 48941 system with the status of the process. The *si_signo* member shall always be equal to SIGCHLD.

48942 RETURN VALUE

48943 If WNOHANG was specified and there are no children to wait for, 0 shall be returned. If *waitid()*
 48944 returns due to the change of state of one of its children, 0 shall be returned. Otherwise, -1 shall
 48945 be returned and *errno* set to indicate the error.

48946 ERRORS

48947 The *waitid()* function shall fail if:

48948 [ECHILD] The calling process has no existing unwaited-for child processes.

48949 [EINTR] The *waitid()* function was interrupted by a signal.

48950 [EINVAL] An invalid value was specified for *options*, or *idtype* and *id* specify an invalid
 48951 set of processes.

48952 **EXAMPLES**

48953 None.

48954 **APPLICATION USAGE**

48955 None.

48956 **RATIONALE**

48957 None.

48958 **FUTURE DIRECTIONS**

48959 None.

48960 **SEE ALSO**48961 *exec*, *exit()*, *wait()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/wait.h>48962 **CHANGE HISTORY**

48963 First released in Issue 4, Version 2.

48964 **Issue 5**

48965 Moved from X/OPEN UNIX extension to BASE.

48966 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

48967 **Issue 6**

48968 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

48969 NAME

48970 waitpid — wait for a child process to stop or terminate

48971 SYNOPSIS

48972 #include <sys/wait.h>

48973 pid_t waitpid(pid_t *pid*, int **stat_loc*, int *options*);

48974 DESCRIPTION

48975 Refer to *wait()*.

48976 **NAME**48977 `wrtomb` — convert a wide-character code to a character (restartable)48978 **SYNOPSIS**48979 `#include <stdio.h>`48980 `size_t wrtomb(char *restrict s, wchar_t wc, mbstate_t *restrict ps);`48981 **DESCRIPTION**

48982 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 48983 conflict between the requirements described here and the ISO C standard is unintentional. This
 48984 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

48985 If *s* is a null pointer, the `wrtomb()` function shall be equivalent to the call:48986 `wrtomb(buf, L'\0', ps)`48987 where *buf* is an internal buffer.

48988 If *s* is not a null pointer, the `wrtomb()` function shall determine the number of bytes needed to
 48989 represent the character that corresponds to the wide character given by *wc* (including any shift
 48990 sequences), and store the resulting bytes in the array whose first element is pointed to by *s*. At
 48991 most {MB_CUR_MAX} bytes are stored. If *wc* is a null wide character, a null byte shall be stored,
 48992 preceded by any shift sequence needed to restore the initial shift state. The resulting state
 48993 described shall be the initial conversion state.

48994 If *ps* is a null pointer, the `wrtomb()` function shall use its own internal **mbstate_t** object, which is
 48995 initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t** object
 48996 pointed to by *ps* shall be used to completely describe the current conversion state of the
 48997 associated character sequence. The implementation shall behave as if no function defined in this
 48998 volume of IEEE Std 1003.1-2001 calls `wrtomb()`.

48999 CX If the application uses any of the `_POSIX_THREAD_SAFE_FUNCTIONS` or `_POSIX_THREADS`
 49000 functions, the application shall ensure that the `wrtomb()` function is called with a non-NULL *ps*
 49001 argument.

49002 The behavior of this function shall be affected by the `LC_CTYPE` category of the current locale.49003 **RETURN VALUE**

49004 The `wrtomb()` function shall return the number of bytes stored in the array object (including any
 49005 shift sequences). When *wc* is not a valid wide character, an encoding error shall occur. In this
 49006 case, the function shall store the value of the macro [EILSEQ] in *errno* and shall return (**size_t**)−1;
 49007 the conversion state shall be undefined.

49008 **ERRORS**49009 The `wrtomb()` function may fail if:49010 CX [EINVAL] *ps* points to an object that contains an invalid conversion state.

49011 [EILSEQ] Invalid wide-character code is detected.

49012 EXAMPLES

49013 None.

49014 APPLICATION USAGE

49015 None.

49016 RATIONALE

49017 None.

49018 FUTURE DIRECTIONS

49019 None.

49020 SEE ALSO

49021 *mbstinit()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**wchar.h**>

49022 CHANGE HISTORY

49023 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

49025 Issue 6

49026 In the DESCRIPTION, a note on using this function in a threaded application is added.

49027 Extensions beyond the ISO C standard are marked.

49028 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

49029 The *wcrtomb()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

49030 **NAME**

49031 wscat — concatenate two wide-character strings

49032 **SYNOPSIS**

49033 #include <wchar.h>

49034 wchar_t *wscat(wchar_t *restrict ws1, const wchar_t *restrict ws2);

49035 **DESCRIPTION**

49036 cx The functionality described on this reference page is aligned with the ISO C standard. Any
49037 conflict between the requirements described here and the ISO C standard is unintentional. This
49038 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

49039 The `wscat()` function shall append a copy of the wide-character string pointed to by `ws2`
49040 (including the terminating null wide-character code) to the end of the wide-character string
49041 pointed to by `ws1`. The initial wide-character code of `ws2` shall overwrite the null wide-character
49042 code at the end of `ws1`. If copying takes place between objects that overlap, the behavior is
49043 undefined.

49044 **RETURN VALUE**49045 The `wscat()` function shall return `ws1`; no return value is reserved to indicate an error.49046 **ERRORS**

49047 No errors are defined.

49048 **EXAMPLES**

49049 None.

49050 **APPLICATION USAGE**

49051 None.

49052 **RATIONALE**

49053 None.

49054 **FUTURE DIRECTIONS**

49055 None.

49056 **SEE ALSO**49057 `wcsncat()`, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>49058 **CHANGE HISTORY**

49059 First released in Issue 4. Derived from the MSE working draft.

49060 **Issue 6**

49061 The Open Group Corrigendum U040/2 is applied. In the RETURN VALUE section, `s1` is changed
49062 to `ws1`.

49063 The `wscat()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

49064 **NAME**

49065 wchr — wide-character string scanning operation

49066 **SYNOPSIS**

49067 #include <wchr.h>

49068 wchr_t *wchr(const wchr_t *ws, wchr_t wc);

49069 **DESCRIPTION**

49070 cx The functionality described on this reference page is aligned with the ISO C standard. Any
49071 conflict between the requirements described here and the ISO C standard is unintentional. This
49072 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

49073 The *wchr()* function shall locate the first occurrence of *wc* in the wide-character string pointed
49074 to by *ws*. The application shall ensure that the value of *wc* is a character representable as a type
49075 **wchr_t** and a wide-character code corresponding to a valid character in the current locale. The
49076 terminating null wide-character code is considered to be part of the wide-character string.

49077 **RETURN VALUE**

49078 Upon completion, *wchr()* shall return a pointer to the wide-character code, or a null pointer if
49079 the wide-character code is not found.

49080 **ERRORS**

49081 No errors are defined.

49082 **EXAMPLES**

49083 None.

49084 **APPLICATION USAGE**

49085 None.

49086 **RATIONALE**

49087 None.

49088 **FUTURE DIRECTIONS**

49089 None.

49090 **SEE ALSO**49091 *wchr()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchr.h>49092 **CHANGE HISTORY**

49093 First released in Issue 4. Derived from the MSE working draft.

49094 **Issue 6**

49095 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

49096 **NAME**

49097 wcscmp — compare two wide-character strings

49098 **SYNOPSIS**

49099 #include <wchar.h>

49100 int wcscmp(const wchar_t *ws1, const wchar_t *ws2);

49101 **DESCRIPTION**

49102 cx The functionality described on this reference page is aligned with the ISO C standard. Any
49103 conflict between the requirements described here and the ISO C standard is unintentional. This
49104 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

49105 The *wcscmp()* function shall compare the wide-character string pointed to by *ws1* to the wide-
49106 character string pointed to by *ws2*.

49107 The sign of a non-zero return value shall be determined by the sign of the difference between the
49108 values of the first pair of wide-character codes that differ in the objects being compared.

49109 **RETURN VALUE**

49110 Upon completion, *wcscmp()* shall return an integer greater than, equal to, or less than 0, if the
49111 wide-character string pointed to by *ws1* is greater than, equal to, or less than the wide-character
49112 string pointed to by *ws2*, respectively.

49113 **ERRORS**

49114 No errors are defined.

49115 **EXAMPLES**

49116 None.

49117 **APPLICATION USAGE**

49118 None.

49119 **RATIONALE**

49120 None.

49121 **FUTURE DIRECTIONS**

49122 None.

49123 **SEE ALSO**49124 *wcsncmp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>49125 **CHANGE HISTORY**

49126 First released in Issue 4. Derived from the MSE working draft.

49127 **NAME**

49128 wscoll — wide-character string comparison using collating information

49129 **SYNOPSIS**

49130 #include <wchar.h>

49131 int wscoll(const wchar_t *ws1, const wchar_t *ws2);

49132 **DESCRIPTION**

49133 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 49134 conflict between the requirements described here and the ISO C standard is unintentional. This
 49135 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

49136 The *wscoll()* function shall compare the wide-character string pointed to by *ws1* to the wide-
 49137 character string pointed to by *ws2*, both interpreted as appropriate to the *LC_COLLATE* category
 49138 of the current locale.

49139 CX The *wscoll()* function shall not change the setting of *errno* if successful.

49140 An application wishing to check for error situations should set *errno* to 0 before calling *wscoll()*.
 49141 If *errno* is non-zero on return, an error has occurred.

49142 **RETURN VALUE**

49143 Upon successful completion, *wscoll()* shall return an integer greater than, equal to, or less than
 49144 0, according to whether the wide-character string pointed to by *ws1* is greater than, equal to, or
 49145 less than the wide-character string pointed to by *ws2*, when both are interpreted as appropriate
 49146 CX to the current locale. On error, *wscoll()* shall set *errno*, but no return value is reserved to
 49147 indicate an error.

49148 **ERRORS**49149 The *wscoll()* function may fail if:

49150 CX [EINVAL] The *ws1* or *ws2* arguments contain wide-character codes outside the domain of
 49151 the collating sequence.

49152 **EXAMPLES**

49153 None.

49154 **APPLICATION USAGE**49155 The *wcsxfrm()* and *wscmp()* functions should be used for sorting large lists.49156 **RATIONALE**

49157 None.

49158 **FUTURE DIRECTIONS**

49159 None.

49160 **SEE ALSO**49161 *wscmp()*, *wcsxfrm()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>49162 **CHANGE HISTORY**

49163 First released in Issue 4. Derived from the MSE working draft.

49164 **Issue 5**

49165 Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

49166 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

49167 **NAME**

49168 wcscpy — copy a wide-character string

49169 **SYNOPSIS**

49170 #include <wchar.h>

49171 wchar_t *wcscpy(wchar_t *restrict ws1, const wchar_t *restrict ws2);

49172 **DESCRIPTION**

49173 cx The functionality described on this reference page is aligned with the ISO C standard. Any
49174 conflict between the requirements described here and the ISO C standard is unintentional. This
49175 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

49176 The *wcscpy()* function shall copy the wide-character string pointed to by *ws2* (including the
49177 terminating null wide-character code) into the array pointed to by *ws1*. If copying takes place
49178 between objects that overlap, the behavior is undefined.

49179 **RETURN VALUE**49180 The *wcscpy()* function shall return *ws1*; no return value is reserved to indicate an error.49181 **ERRORS**

49182 No errors are defined.

49183 **EXAMPLES**

49184 None.

49185 **APPLICATION USAGE**

49186 None.

49187 **RATIONALE**

49188 None.

49189 **FUTURE DIRECTIONS**

49190 None.

49191 **SEE ALSO**49192 *wcsncpy()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>49193 **CHANGE HISTORY**

49194 First released in Issue 4. Derived from the MSE working draft.

49195 **Issue 6**49196 The *wcscpy()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

49197 **NAME**

49198 wcscspn — get the length of a complementary wide substring

49199 **SYNOPSIS**

49200 #include <wchar.h>

49201 size_t wcscspn(const wchar_t *ws1, const wchar_t *ws2);

49202 **DESCRIPTION**

49203 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
49204 conflict between the requirements described here and the ISO C standard is unintentional. This
49205 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

49206 The *wcscspn()* function shall compute the length (in wide characters) of the maximum initial
49207 segment of the wide-character string pointed to by *ws1* which consists entirely of wide-character
49208 codes *not* from the wide-character string pointed to by *ws2*.

49209 **RETURN VALUE**

49210 The *wcscspn()* function shall return the length of the initial substring of *ws1*; no return value is
49211 reserved to indicate an error.

49212 **ERRORS**

49213 No errors are defined.

49214 **EXAMPLES**

49215 None.

49216 **APPLICATION USAGE**

49217 None.

49218 **RATIONALE**

49219 None.

49220 **FUTURE DIRECTIONS**

49221 None.

49222 **SEE ALSO**49223 *wcspn()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>49224 **CHANGE HISTORY**

49225 First released in Issue 4. Derived from the MSE working draft.

49226 **Issue 5**

49227 The RETURN VALUE section is updated to indicate that *wcscspn()* returns the length of *ws1*,
49228 rather than *ws1* itself.

49229 **NAME**49230 `wcsftime` — convert date and time to a wide-character string49231 **SYNOPSIS**49232 `#include <wchar.h>`

49233 `size_t wcsftime(wchar_t *restrict wcs, size_t maxsize,`
 49234 `const wchar_t *restrict format, const struct tm *restrict timeptr);`

49235 **DESCRIPTION**

49236 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 49237 conflict between the requirements described here and the ISO C standard is unintentional. This
 49238 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

49239 The `wcsftime()` function shall be equivalent to the `strftime()` function, except that:

- 49240 • The argument `wcs` points to the initial element of an array of wide characters into which the
 49241 generated output is to be placed.
- 49242 • The argument `maxsize` indicates the maximum number of wide characters to be placed in the
 49243 output array.
- 49244 • The argument `format` is a wide-character string and the conversion specifications are replaced
 49245 by corresponding sequences of wide characters.
- 49246 • The return value indicates the number of wide characters placed in the output array.

49247 If copying takes place between objects that overlap, the behavior is undefined.

49248 **RETURN VALUE**

49249 If the total number of resulting wide-character codes including the terminating null wide-
 49250 character code is no more than `maxsize`, `wcsftime()` shall return the number of wide-character
 49251 codes placed into the array pointed to by `wcs`, not including the terminating null wide-character
 49252 code. Otherwise, zero is returned and the contents of the array are unspecified.

49253 **ERRORS**

49254 No errors are defined.

49255 **EXAMPLES**

49256 None.

49257 **APPLICATION USAGE**

49258 None.

49259 **RATIONALE**

49260 None.

49261 **FUTURE DIRECTIONS**

49262 None.

49263 **SEE ALSO**49264 `strftime()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<wchar.h>`49265 **CHANGE HISTORY**

49266 First released in Issue 4.

49267 **Issue 5**

49268 Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

49269 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of the `format`
 49270 argument is changed from `const char *` to `const wchar_t *`.

49271 **Issue 6**

49272 The *wcsftime()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

49273 **NAME**

49274 wcslen — get wide-character string length

49275 **SYNOPSIS**

49276 #include <wchar.h>

49277 size_t wcslen(const wchar_t *ws);

49278 **DESCRIPTION**

49279 cx The functionality described on this reference page is aligned with the ISO C standard. Any
49280 conflict between the requirements described here and the ISO C standard is unintentional. This
49281 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

49282 The *wcslen()* function shall compute the number of wide-character codes in the wide-character
49283 string to which *ws* points, not including the terminating null wide-character code.

49284 **RETURN VALUE**

49285 The *wcslen()* function shall return the length of *ws*; no return value is reserved to indicate an
49286 error.

49287 **ERRORS**

49288 No errors are defined.

49289 **EXAMPLES**

49290 None.

49291 **APPLICATION USAGE**

49292 None.

49293 **RATIONALE**

49294 None.

49295 **FUTURE DIRECTIONS**

49296 None.

49297 **SEE ALSO**49298 The Base Definitions volume of IEEE Std 1003.1-2001, <**wchar.h**>49299 **CHANGE HISTORY**

49300 First released in Issue 4. Derived from the MSE working draft.

49301 **NAME**

49302 wcsncat — concatenate a wide-character string with part of another

49303 **SYNOPSIS**

49304 #include <wchar.h>

49305 wchar_t *wcsncat(wchar_t *restrict ws1, const wchar_t *restrict ws2,
49306 size_t n);49307 **DESCRIPTION**49308 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
49309 conflict between the requirements described here and the ISO C standard is unintentional. This
49310 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.49311 The *wcsncat()* function shall append not more than *n* wide-character codes (a null wide-
49312 character code and wide-character codes that follow it are not appended) from the array pointed
49313 to by *ws2* to the end of the wide-character string pointed to by *ws1*. The initial wide-character
49314 code of *ws2* shall overwrite the null wide-character code at the end of *ws1*. A terminating null
49315 wide-character code shall always be appended to the result. If copying takes place between
49316 objects that overlap, the behavior is undefined.49317 **RETURN VALUE**49318 The *wcsncat()* function shall return *ws1*; no return value is reserved to indicate an error.49319 **ERRORS**

49320 No errors are defined.

49321 **EXAMPLES**

49322 None.

49323 **APPLICATION USAGE**

49324 None.

49325 **RATIONALE**

49326 None.

49327 **FUTURE DIRECTIONS**

49328 None.

49329 **SEE ALSO**49330 *wscat()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>49331 **CHANGE HISTORY**

49332 First released in Issue 4. Derived from the MSE working draft.

49333 **Issue 6**49334 The *wcsncat()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

49335 **NAME**49336 `wcsncmp` — compare part of two wide-character strings49337 **SYNOPSIS**49338 `#include <wchar.h>`49339 `int wcsncmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);`49340 **DESCRIPTION**

49341 cx The functionality described on this reference page is aligned with the ISO C standard. Any
49342 conflict between the requirements described here and the ISO C standard is unintentional. This
49343 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

49344 The `wcsncmp()` function shall compare not more than *n* wide-character codes (wide-character
49345 codes that follow a null wide-character code are not compared) from the array pointed to by *ws1*
49346 to the array pointed to by *ws2*.

49347 The sign of a non-zero return value shall be determined by the sign of the difference between the
49348 values of the first pair of wide-character codes that differ in the objects being compared.

49349 **RETURN VALUE**

49350 Upon successful completion, `wcsncmp()` shall return an integer greater than, equal to, or less
49351 than 0, if the possibly null-terminated array pointed to by *ws1* is greater than, equal to, or less
49352 than the possibly null-terminated array pointed to by *ws2*, respectively.

49353 **ERRORS**

49354 No errors are defined.

49355 **EXAMPLES**

49356 None.

49357 **APPLICATION USAGE**

49358 None.

49359 **RATIONALE**

49360 None.

49361 **FUTURE DIRECTIONS**

49362 None.

49363 **SEE ALSO**49364 `wscmp()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<wchar.h>`49365 **CHANGE HISTORY**

49366 First released in Issue 4. Derived from the MSE working draft.

49367 **NAME**

49368 wcsncpy — copy part of a wide-character string

49369 **SYNOPSIS**

49370 #include <wchar.h>

49371 wchar_t *wcsncpy(wchar_t *restrict ws1, const wchar_t *restrict ws2,
49372 size_t n);49373 **DESCRIPTION**49374 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
49375 conflict between the requirements described here and the ISO C standard is unintentional. This
49376 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.49377 The *wcsncpy()* function shall copy not more than *n* wide-character codes (wide-character codes
49378 that follow a null wide-character code are not copied) from the array pointed to by *ws2* to the
49379 array pointed to by *ws1*. If copying takes place between objects that overlap, the behavior is
49380 undefined.49381 If the array pointed to by *ws2* is a wide-character string that is shorter than *n* wide-character
49382 codes, null wide-character codes shall be appended to the copy in the array pointed to by *ws1*,
49383 until *n* wide-character codes in all are written.49384 **RETURN VALUE**49385 The *wcsncpy()* function shall return *ws1*; no return value is reserved to indicate an error.49386 **ERRORS**

49387 No errors are defined.

49388 **EXAMPLES**

49389 None.

49390 **APPLICATION USAGE**49391 If there is no null wide-character code in the first *n* wide-character codes of the array pointed to
49392 by *ws2*, the result is not null-terminated.49393 **RATIONALE**

49394 None.

49395 **FUTURE DIRECTIONS**

49396 None.

49397 **SEE ALSO**49398 *wscpy()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>49399 **CHANGE HISTORY**

49400 First released in Issue 4. Derived from the MSE working draft.

49401 **Issue 6**49402 The *wcsncpy()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

49403 **NAME**

49404 wcsprk — scan a wide-character string for a wide-character code

49405 **SYNOPSIS**

49406 #include <wchar.h>

49407 wchar_t *wcsprk(const wchar_t *ws1, const wchar_t *ws2);

49408 **DESCRIPTION**

49409 cx The functionality described on this reference page is aligned with the ISO C standard. Any
49410 conflict between the requirements described here and the ISO C standard is unintentional. This
49411 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

49412 The *wcsprk()* function shall locate the first occurrence in the wide-character string pointed to by
49413 *ws1* of any wide-character code from the wide-character string pointed to by *ws2*.

49414 **RETURN VALUE**

49415 Upon successful completion, *wcsprk()* shall return a pointer to the wide-character code or a null
49416 pointer if no wide-character code from *ws2* occurs in *ws1*.

49417 **ERRORS**

49418 No errors are defined.

49419 **EXAMPLES**

49420 None.

49421 **APPLICATION USAGE**

49422 None.

49423 **RATIONALE**

49424 None.

49425 **FUTURE DIRECTIONS**

49426 None.

49427 **SEE ALSO**49428 *wchr()*, *wchr()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>49429 **CHANGE HISTORY**

49430 First released in Issue 4. Derived from the MSE working draft.

49431 **NAME**

49432 wcsrchr — wide-character string scanning operation

49433 **SYNOPSIS**

49434 #include <wchar.h>

49435 wchar_t *wcsrchr(const wchar_t *ws, wchar_t wc);

49436 **DESCRIPTION**

49437 cx The functionality described on this reference page is aligned with the ISO C standard. Any
49438 conflict between the requirements described here and the ISO C standard is unintentional. This
49439 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

49440 The *wcsrchr()* function shall locate the last occurrence of *wc* in the wide-character string pointed
49441 to by *ws*. The application shall ensure that the value of *wc* is a character representable as a type
49442 **wchar_t** and a wide-character code corresponding to a valid character in the current locale. The
49443 terminating null wide-character code shall be considered to be part of the wide-character string.

49444 **RETURN VALUE**

49445 Upon successful completion, *wcsrchr()* shall return a pointer to the wide-character code or a null
49446 pointer if *wc* does not occur in the wide-character string.

49447 **ERRORS**

49448 No errors are defined.

49449 **EXAMPLES**

49450 None.

49451 **APPLICATION USAGE**

49452 None.

49453 **RATIONALE**

49454 None.

49455 **FUTURE DIRECTIONS**

49456 None.

49457 **SEE ALSO**49458 *wcschr()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**wchar.h**>49459 **CHANGE HISTORY**

49460 First released in Issue 4. Derived from the MSE working draft.

49461 **Issue 6**

49462 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

49463 NAME

49464 `wcsrtombs` — convert a wide-character string to a character string (restartable)

49465 SYNOPSIS

49466 `#include <wchar.h>`

```
49467        size_t wcsrtombs(char *restrict dst, const wchar_t **restrict src,  

49468            size_t len, mbstate_t *restrict ps);
```

49469 DESCRIPTION

49470 CX The functionality described on this reference page is aligned with the ISO C standard. Any
49471 conflict between the requirements described here and the ISO C standard is unintentional. This
49472 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

49473 The `wcsrtombs()` function shall convert a sequence of wide characters from the array indirectly
49474 pointed to by `src` into a sequence of corresponding characters, beginning in the conversion state
49475 described by the object pointed to by `ps`. If `dst` is not a null pointer, the converted characters
49476 shall then be stored into the array pointed to by `dst`. Conversion continues up to and including a
49477 terminating null wide character, which shall also be stored. Conversion shall stop earlier in the
49478 following cases:

- 49479 • When a code is reached that does not correspond to a valid character
- 49480 • When the next character would exceed the limit of `len` total bytes to be stored in the array
- 49481 pointed to by `dst` (and `dst` is not a null pointer)

49482 Each conversion shall take place as if by a call to the `wcrtomb()` function.

49483 If `dst` is not a null pointer, the pointer object pointed to by `src` shall be assigned either a null
49484 pointer (if conversion stopped due to reaching a terminating null wide character) or the address
49485 just past the last wide character converted (if any). If conversion stopped due to reaching a
49486 terminating null wide character, the resulting state described shall be the initial conversion state.

49487 If `ps` is a null pointer, the `wcsrtombs()` function shall use its own internal `mbstate_t` object, which
49488 is initialized at program start-up to the initial conversion state. Otherwise, the `mbstate_t` object
49489 pointed to by `ps` shall be used to completely describe the current conversion state of the
49490 associated character sequence. The implementation shall behave as if no function defined in this
49491 volume of IEEE Std 1003.1-2001 calls `wcsrtombs()`.

49492 CX If the application uses any of the `_POSIX_THREAD_SAFE_FUNCTIONS` or `_POSIX_THREADS`
49493 functions, the application shall ensure that the `wcsrtombs()` function is called with a non-NULL
49494 `ps` argument.

49495 The behavior of this function shall be affected by the `LC_CTYPE` category of the current locale.

49496 RETURN VALUE

49497 If conversion stops because a code is reached that does not correspond to a valid character, an
49498 encoding error occurs. In this case, the `wcsrtombs()` function shall store the value of the macro
49499 `[EILSEQ]` in `errno` and return `(size_t)-1`; the conversion state is undefined. Otherwise, it shall
49500 return the number of bytes in the resulting character sequence, not including the terminating
49501 null (if any).

49502 ERRORS

49503 The `wcsrtombs()` function may fail if:

- 49504 CX `[EINVAL]` `ps` points to an object that contains an invalid conversion state.
- 49505 `[EILSEQ]` A wide-character code does not correspond to a valid character.

49506 **EXAMPLES**

49507 None.

49508 **APPLICATION USAGE**

49509 None.

49510 **RATIONALE**

49511 None.

49512 **FUTURE DIRECTIONS**

49513 None.

49514 **SEE ALSO**49515 *mbstinit()*, *wcrtomb()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>49516 **CHANGE HISTORY**49517 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
49518 (E).49519 **Issue 6**

49520 In the DESCRIPTION, a note on using this function in a threaded application is added.

49521 Extensions beyond the ISO C standard are marked.

49522 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

49523 The *wcsrtoombs()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

49524 **NAME**

49525 wcsspnp — get the length of a wide substring

49526 **SYNOPSIS**

49527 #include <wchar.h>

49528 size_t wcsspnp(const wchar_t *ws1, const wchar_t *ws2);

49529 **DESCRIPTION**

49530 cx The functionality described on this reference page is aligned with the ISO C standard. Any
49531 conflict between the requirements described here and the ISO C standard is unintentional. This
49532 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

49533 The wcsspnp() function shall compute the length (in wide characters) of the maximum initial
49534 segment of the wide-character string pointed to by ws1 which consists entirely of wide-character
49535 codes from the wide-character string pointed to by ws2.

49536 **RETURN VALUE**

49537 The wcsspnp() function shall return the length of the initial substring of ws1; no return value is
49538 reserved to indicate an error.

49539 **ERRORS**

49540 No errors are defined.

49541 **EXAMPLES**

49542 None.

49543 **APPLICATION USAGE**

49544 None.

49545 **RATIONALE**

49546 None.

49547 **FUTURE DIRECTIONS**

49548 None.

49549 **SEE ALSO**

49550 wcscspnp(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

49551 **CHANGE HISTORY**

49552 First released in Issue 4. Derived from the MSE working draft.

49553 **Issue 5**

49554 The RETURN VALUE section is updated to indicate that wcsspnp() returns the length of ws1
49555 rather than ws1 itself.

49556 **NAME**

49557 wcsstr — find a wide-character substring

49558 **SYNOPSIS**

49559 #include <wchar.h>

49560 wchar_t *wcsstr(const wchar_t *restrict ws1,
49561 const wchar_t *restrict ws2);49562 **DESCRIPTION**49563 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
49564 conflict between the requirements described here and the ISO C standard is unintentional. This
49565 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.49566 The *wcsstr()* function shall locate the first occurrence in the wide-character string pointed to by
49567 *ws1* of the sequence of wide characters (excluding the terminating null wide character) in the
49568 wide-character string pointed to by *ws2*.49569 **RETURN VALUE**49570 Upon successful completion, *wcsstr()* shall return a pointer to the located wide-character string,
49571 or a null pointer if the wide-character string is not found.49572 If *ws2* points to a wide-character string with zero length, the function shall return *ws1*.49573 **ERRORS**

49574 No errors are defined.

49575 **EXAMPLES**

49576 None.

49577 **APPLICATION USAGE**

49578 None.

49579 **RATIONALE**

49580 None.

49581 **FUTURE DIRECTIONS**

49582 None.

49583 **SEE ALSO**49584 *wcschr()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>49585 **CHANGE HISTORY**49586 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
49587 (E).49588 **Issue 6**49589 The *wcsstr()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

49590 NAME

49591 wcstod, wcstof, wcstold — convert a wide-character string to a double-precision number

49592 SYNOPSIS

49593 #include <wchar.h>

49594 double wcstod(const wchar_t *restrict nptr, wchar_t **restrict endptr);

49595 float wcstof(const wchar_t *restrict nptr, wchar_t **restrict endptr);

49596 long double wcstold(const wchar_t *restrict nptr,

49597 wchar_t **restrict endptr);

49598 DESCRIPTION

49599 cx The functionality described on this reference page is aligned with the ISO C standard. Any
 49600 conflict between the requirements described here and the ISO C standard is unintentional. This
 49601 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

49602 These functions shall convert the initial portion of the wide-character string pointed to by *nptr* to
 49603 **double**, **float**, and **long double** representation, respectively. First, they shall decompose the
 49604 input wide-character string into three parts:

- 49605 1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by
 49606 *iswspace()*)
- 49607 2. A subject sequence interpreted as a floating-point constant or representing infinity or NaN
- 49608 3. A final wide-character string of one or more unrecognized wide-character codes, including
 49609 the terminating null wide-character code of the input wide-character string

49610 Then they shall attempt to convert the subject sequence to a floating-point number, and return
 49611 the result.

49612 The expected form of the subject sequence is an optional plus or minus sign, then one of the
 49613 following:

- 49614 • A non-empty sequence of decimal digits optionally containing a radix character, then an
 49615 optional exponent part
- 49616 • A 0x or 0X, then a non-empty sequence of hexadecimal digits optionally containing a radix
 49617 character, then an optional binary exponent part
- 49618 • One of INF or INFINITY, or any other wide string equivalent except for case
- 49619 • One of NAN or NAN(*n-wchar-sequence_{opt}*), or any other wide string ignoring case in the NAN
 49620 part, where:

49621 n-wchar-sequence:

49622 digit

49623 nondigit

49624 n-wchar-sequence digit

49625 n-wchar-sequence nondigit

49626 The subject sequence is defined as the longest initial subsequence of the input wide string,
 49627 starting with the first non-white-space wide character, that is of the expected form. The subject
 49628 sequence contains no wide characters if the input wide string is not of the expected form.

49629 If the subject sequence has the expected form for a floating-point number, the sequence of wide
 49630 characters starting with the first digit or the radix character (whichever occurs first) shall be
 49631 interpreted as a floating constant according to the rules of the C language, except that the radix
 49632 character shall be used in place of a period, and that if neither an exponent part nor a radix
 49633 character appears in a decimal floating-point number, or if a binary exponent part does not

49634 appear in a hexadecimal floating-point number, an exponent part of the appropriate type with
 49635 value zero shall be assumed to follow the last digit in the string. If the subject sequence begins
 49636 with a minus sign, the sequence shall be interpreted as negated. A wide-character sequence INF
 49637 or INFINITY shall be interpreted as an infinity, if representable in the return type, else as if it
 49638 were a floating constant that is too large for the range of the return type. A wide-character
 49639 sequence NAN or NAN(*n-wchar-sequence_{opt}*) shall be interpreted as a quiet NaN, if supported in
 49640 the return type, else as if it were a subject sequence part that does not have the expected form;
 49641 the meaning of the *n-wchar* sequences is implementation-defined. A pointer to the final wide
 49642 string shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

49643 If the subject sequence has the hexadecimal form and FLT_RADIX is a power of 2, the
 49644 conversion shall be rounded in an implementation-defined manner.

49645 CX The radix character shall be as defined in the program's locale (category *LC_NUMERIC*). In the
 49646 POSIX locale, or in a locale where the radix character is not defined, the radix character shall
 49647 default to a period (' . ').

49648 CX In other than the C or POSIX locales, other implementation-defined subject sequences may be
 49649 accepted.

49650 If the subject sequence is empty or does not have the expected form, no conversion shall be
 49651 performed; the value of *nptr* shall be stored in the object pointed to by *endptr*, provided that
 49652 *endptr* is not a null pointer.

49653 CX The *wcstod()* function shall not change the setting of *errno* if successful.

49654 Since 0 is returned on error and is also a valid return on success, an application wishing to check
 49655 for error situations should set *errno* to 0, then call *wcstod()*, *wcstof()*, or *wcstold()*, then check
 49656 *errno*.

49657 **RETURN VALUE**

49658 Upon successful completion, these functions shall return the converted value. If no conversion
 49659 CX could be performed, 0 shall be returned and *errno* may be set to [EINVAL].

49660 If the correct value is outside the range of representable values, plus or minus HUGE_VAL,
 49661 HUGE_VALF, or HUGE_VALL shall be returned (according to the sign of the value), and *errno*
 49662 shall be set to [ERANGE].

49663 If the correct value would cause underflow, a value whose magnitude is no greater than the
 49664 smallest normalized positive number in the return type shall be returned and *errno* set to
 49665 [ERANGE].

49666 **ERRORS**

49667 The *wcstod()* function shall fail if:

49668 [ERANGE] The value to be returned would cause overflow or underflow.

49669 The *wcstod()* function may fail if:

49670 CX [EINVAL] No conversion could be performed.

49671 **EXAMPLES**

49672 None.

49673 **APPLICATION USAGE**

49674 If the subject sequence has the hexadecimal form and FLT_RADIX is not a power of 2, and the
 49675 result is not exactly representable, the result should be one of the two numbers in the
 49676 appropriate internal format that are adjacent to the hexadecimal floating source value, with the
 49677 extra stipulation that the error should have a correct sign for the current rounding direction.

49678 If the subject sequence has the decimal form and at most DECIMAL_DIG (defined in <float.h>)
 49679 significant digits, the result should be correctly rounded. If the subject sequence *D* has the
 49680 decimal form and more than DECIMAL_DIG significant digits, consider the two bounding,
 49681 adjacent decimal strings *L* and *U*, both having DECIMAL_DIG significant digits, such that the
 49682 values of *L*, *D*, and *U* satisfy "*L* <= *D* <= *U*". The result should be one of the (equal or
 49683 adjacent) values that would be obtained by correctly rounding *L* and *U* according to the current
 49684 rounding direction, with the extra stipulation that the error with respect to *D* should have a
 49685 correct sign for the current rounding direction.

49686 **RATIONALE**

49687 None.

49688 **FUTURE DIRECTIONS**

49689 None.

49690 **SEE ALSO**

49691 *iswspace()*, *localeconv()*, *scanf()*, *setlocale()*, *wcstol()*, the Base Definitions volume of
 49692 IEEE Std 1003.1-2001, Chapter 7, Locale, <float.h>, <wchar.h>

49693 **CHANGE HISTORY**

49694 First released in Issue 4. Derived from the MSE working draft.

49695 **Issue 5**49696 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.49697 **Issue 6**

49698 Extensions beyond the ISO C standard are marked.

49699 The following new requirements on POSIX implementations derive from alignment with the
 49700 Single UNIX Specification:

- 49701 • In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
- 49702 added if no conversion could be performed.

49703 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 49704 • The *wcstod()* prototype is updated.
- 49705 • The *wcstof()* and *wcstold()* functions are added.
- 49706 • If the correct value for *wcstod()* would cause underflow, the return value changed from 0 (as
- 49707 specified in Issue 5) to the smallest normalized positive number.
- 49708 • The DESCRIPTION, RETURN VALUE, and APPLICATION USAGE sections are extensively
- 49709 updated.

49710 ISO/IEC 9899:1999 standard, Technical Corrigendum No. 1 is incorporated.

49711 **NAME**

49712 wcstoimax, wcstoumax — convert a wide-character string to an integer type

49713 **SYNOPSIS**

49714 #include <stddef.h>

49715 #include <inttypes.h>

49716 intmax_t wcstoimax(const wchar_t *restrict nptr,

49717 wchar_t **restrict endptr, int base);

49718 uintmax_t wcstoumax(const wchar_t *restrict nptr,

49719 wchar_t **restrict endptr, int base);

49720 **DESCRIPTION**

49721 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
 49722 conflict between the requirements described here and the ISO C standard is unintentional. This
 49723 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

49724 These functions shall be equivalent to the *wcstol()*, *wcstoll()*, *wcstoul()*, and *wcstoull()* functions,
 49725 respectively, except that the initial portion of the wide string shall be converted to **intmax_t** and
 49726 **uintmax_t** representation, respectively.

49727 **RETURN VALUE**

49728 These functions shall return the converted value, if any.

49729 If no conversion could be performed, zero shall be returned. If the correct value is outside the
 49730 range of representable values, {INTMAX_MAX}, {INTMAX_MIN}, or {UINTMAX_MAX} shall
 49731 be returned (according to the return type and sign of the value, if any), and *errno* shall be set to
 49732 [ERANGE].

49733 **ERRORS**

49734 These functions shall fail if:

49735 [EINVAL] The value of *base* is not supported.

49736 [ERANGE] The value to be returned is not representable.

49737 These functions may fail if:

49738 [EINVAL] No conversion could be performed.

49739 **EXAMPLES**

49740 None.

49741 **APPLICATION USAGE**

49742 None.

49743 **RATIONALE**

49744 None.

49745 **FUTURE DIRECTIONS**

49746 None.

49747 **SEE ALSO**49748 *wcstol()*, *wcstoul()*, the Base Definitions volume of IEEE Std 1003.1-2001, <inttypes.h>,
 49749 <stddef.h>49750 **CHANGE HISTORY**

49751 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

49752 **NAME**

49753 wcstok — split a wide-character string into tokens

49754 **SYNOPSIS**

49755 #include <wchar.h>

49756 wchar_t *wcstok(wchar_t *restrict ws1, const wchar_t *restrict ws2,
49757 wchar_t **restrict ptr);

49758 **DESCRIPTION**

49759 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
49760 conflict between the requirements described here and the ISO C standard is unintentional. This
49761 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

49762 A sequence of calls to *wcstok()* shall break the wide-character string pointed to by *ws1* into a
49763 sequence of tokens, each of which shall be delimited by a wide-character code from the wide-
49764 character string pointed to by *ws2*. The *ptr* argument points to a caller-provided **wchar_t** pointer
49765 into which the *wcstok()* function shall store information necessary for it to continue scanning the
49766 same wide-character string.

49767 The first call in the sequence has *ws1* as its first argument, and is followed by calls with a null
49768 pointer as their first argument. The separator string pointed to by *ws2* may be different from call
49769 to call.

49770 The first call in the sequence shall search the wide-character string pointed to by *ws1* for the first
49771 wide-character code that is *not* contained in the current separator string pointed to by *ws2*. If no
49772 such wide-character code is found, then there are no tokens in the wide-character string pointed
49773 to by *ws1* and *wcstok()* shall return a null pointer. If such a wide-character code is found, it shall
49774 be the start of the first token.

49775 The *wcstok()* function shall then search from there for a wide-character code that *is* contained in
49776 the current separator string. If no such wide-character code is found, the current token extends
49777 to the end of the wide-character string pointed to by *ws1*, and subsequent searches for a token
49778 shall return a null pointer. If such a wide-character code is found, it shall be overwritten by a
49779 null wide character, which terminates the current token. The *wcstok()* function shall save a
49780 pointer to the following wide-character code, from which the next search for a token shall start.

49781 Each subsequent call, with a null pointer as the value of the first argument, shall start searching
49782 from the saved pointer and behave as described above.

49783 The implementation shall behave as if no function calls *wcstok()*.

49784 **RETURN VALUE**

49785 Upon successful completion, the *wcstok()* function shall return a pointer to the first wide-
49786 character code of a token. Otherwise, if there is no token, *wcstok()* shall return a null pointer.

49787 **ERRORS**

49788 No errors are defined.

49789 EXAMPLES

49790 None.

49791 APPLICATION USAGE

49792 None.

49793 RATIONALE

49794 None.

49795 FUTURE DIRECTIONS

49796 None.

49797 SEE ALSO

49798 The Base Definitions volume of IEEE Std 1003.1-2001, <**wchar.h**>

49799 CHANGE HISTORY

49800 First released in Issue 4.

49801 Issue 5

49802 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, a third argument is
49803 added to the definition of *wcstok()* in the SYNOPSIS.

49804 Issue 6

49805 The *wcstok()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

49806 NAME

49807 wcstol, wcstoll — convert a wide-character string to a long integer

49808 SYNOPSIS

49809 #include <wchar.h>

49810 long wcstol(const wchar_t *restrict nptr, wchar_t **restrict endptr,
49811 int base);49812 long long wcstoll(const wchar_t *restrict nptr,
49813 wchar_t **restrict endptr, int base);

49814 DESCRIPTION

49815 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 49816 conflict between the requirements described here and the ISO C standard is unintentional. This
 49817 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

49818 These functions shall convert the initial portion of the wide-character string pointed to by *nptr* to
 49819 **long**, **long long**, **unsigned long**, and **unsigned long long** representation, respectively. First, they
 49820 shall decompose the input string into three parts:

- 49821 1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by
 49822 *iswspace()*)
- 49823 2. A subject sequence interpreted as an integer represented in some radix determined by the
 49824 value of *base*
- 49825 3. A final wide-character string of one or more unrecognized wide-character codes, including
 49826 the terminating null wide-character code of the input wide-character string

49827 Then they shall attempt to convert the subject sequence to an integer, and return the result.

49828 If *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant,
 49829 or hexadecimal constant, any of which may be preceded by a '+' or '-' sign. A decimal
 49830 constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal
 49831 constant consists of the prefix '0' optionally followed by a sequence of the digits '0' to '7'
 49832 only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the
 49833 decimal digits and letters 'a' (or 'A') to 'f' (or 'F') with values 10 to 15 respectively.

49834 If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence
 49835 of letters and digits representing an integer with the radix specified by *base*, optionally preceded
 49836 by a '+' or '-' sign, but not including an integer suffix. The letters from 'a' (or 'A') to 'z'
 49837 (or 'Z') inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less
 49838 than that of *base* shall be permitted. If the value of *base* is 16, the wide-character code
 49839 representations of 0x or 0X may optionally precede the sequence of letters and digits, following
 49840 the sign if present.

49841 The subject sequence is defined as the longest initial subsequence of the input wide-character
 49842 string, starting with the first non-white-space wide-character code that is of the expected form.
 49843 The subject sequence contains no wide-character codes if the input wide-character string is
 49844 empty or consists entirely of white-space wide-character code, or if the first non-white-space
 49845 wide-character code is other than a sign or a permissible letter or digit.

49846 If the subject sequence has the expected form and *base* is 0, the sequence of wide-character codes
 49847 starting with the first digit shall be interpreted as an integer constant. If the subject sequence has
 49848 the expected form and the value of *base* is between 2 and 36, it shall be used as the base for
 49849 conversion, ascribing to each letter its value as given above. If the subject sequence begins with a
 49850 minus sign, the value resulting from the conversion shall be negated. A pointer to the final
 49851 wide-character string shall be stored in the object pointed to by *endptr*, provided that *endptr* is

49852 not a null pointer.

49853 CX In other than the C or POSIX locales, other implementation-defined subject sequences may be
49854 accepted.

49855 If the subject sequence is empty or does not have the expected form, no conversion shall be
49856 performed; the value of *nptr* shall be stored in the object pointed to by *endptr*, provided that
49857 *endptr* is not a null pointer.

49858 CX These functions shall not change the setting of *errno* if successful.

49859 Since 0, {LONG_MIN} or {LLONG_MIN} and {LONG_MAX} or {LLONG_MAX} are returned on
49860 error and are also valid returns on success, an application wishing to check for error situations
49861 should set *errno* to 0, then call *wcstol()* or *wcstoll()*, then check *errno*.

49862 RETURN VALUE

49863 Upon successful completion, these functions shall return the converted value, if any. If no
49864 CX conversion could be performed, 0 shall be returned and *errno* may be set to indicate the error. If
49865 the correct value is outside the range of representable values, {LONG_MIN}, {LONG_MAX},
49866 {LLONG_MIN}, or {LLONG_MAX} shall be returned (according to the sign of the value), and
49867 *errno* set to [ERANGE].

49868 ERRORS

49869 These functions shall fail if:

49870 CX [EINVAL] The value of *base* is not supported.

49871 [ERANGE] The value to be returned is not representable.

49872 These functions may fail if:

49873 CX [EINVAL] No conversion could be performed.

49874 EXAMPLES

49875 None.

49876 APPLICATION USAGE

49877 None.

49878 RATIONALE

49879 None.

49880 FUTURE DIRECTIONS

49881 None.

49882 SEE ALSO

49883 *iswalph()*, *scanf()*, *wcstod()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

49884 CHANGE HISTORY

49885 First released in Issue 4. Derived from the MSE working draft.

49886 Issue 5

49887 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

49888 Issue 6

49889 Extensions beyond the ISO C standard are marked.

49890 The following new requirements on POSIX implementations derive from alignment with the
49891 Single UNIX Specification:

- 49892 • In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
- 49893 added if no conversion could be performed.

- 49894 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:
- 49895 • The *wcstol()* prototype is updated.
- 49896 • The *wcstoll()* function is added.

49897 **NAME**

49898 wcstold — convert a wide-character string to a double-precision number

49899 **SYNOPSIS**

49900 #include <wchar.h>

49901 long double wcstold(const wchar_t *restrict nptr,
49902 wchar_t **restrict endptr);49903 **DESCRIPTION**49904 Refer to *wcstod()*.

49905 **NAME**

49906 wcstoll — convert a wide-character string to a long integer

49907 **SYNOPSIS**

49908 #include <wchar.h>

49909 long long wcstoll(const wchar_t *restrict nptr,

49910 wchar_t **restrict endptr, int base);

49911 **DESCRIPTION**49912 Refer to *wcstol()*.

49913 **NAME**

49914 wcstombs — convert a wide-character string to a character string

49915 **SYNOPSIS**

49916 #include <stdlib.h>

49917 size_t wcstombs(char *restrict s, const wchar_t *restrict pwcs,
49918 size_t n);

49919 **DESCRIPTION**

49920 CX The functionality described on this reference page is aligned with the ISO C standard. Any
49921 conflict between the requirements described here and the ISO C standard is unintentional. This
49922 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

49923 The *wcstombs()* function shall convert the sequence of wide-character codes that are in the array
49924 pointed to by *pwcs* into a sequence of characters that begins in the initial shift state and store
49925 these characters into the array pointed to by *s*, stopping if a character would exceed the limit of *n*
49926 total bytes or if a null byte is stored. Each wide-character code shall be converted as if by a call to
49927 *wctomb()*, except that the shift state of *wctomb()* shall not be affected.

49928 The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

49929 No more than *n* bytes shall be modified in the array pointed to by *s*. If copying takes place
49930 CX between objects that overlap, the behavior is undefined. If *s* is a null pointer, *wcstombs()* shall
49931 return the length required to convert the entire array regardless of the value of *n*, but no values
49932 are stored.

49933 The *wcstombs()* function need not be reentrant. A function that is not required to be reentrant is
49934 not required to be thread-safe.

49935 **RETURN VALUE**

49936 If a wide-character code is encountered that does not correspond to a valid character (of one or
49937 more bytes each), *wcstombs()* shall return (**size_t**)−1. Otherwise, *wcstombs()* shall return the
49938 number of bytes stored in the character array, not including any terminating null byte. The array
49939 shall not be null-terminated if the value returned is *n*.

49940 **ERRORS**

49941 The *wcstombs()* function may fail if:

49942 CX [EILSEQ] A wide-character code does not correspond to a valid character.

49943 **EXAMPLES**

49944 None.

49945 **APPLICATION USAGE**

49946 None.

49947 **RATIONALE**

49948 None.

49949 **FUTURE DIRECTIONS**

49950 None.

49951 **SEE ALSO**

49952 *mblen()*, *mbtowc()*, *mbstowcs()*, *wctomb()*, the Base Definitions volume of IEEE Std 1003.1-2001,
49953 <stdlib.h>

49954 **CHANGE HISTORY**

49955 First released in Issue 4. Derived from the ISO C standard.

49956 **Issue 6**

49957 The following new requirements on POSIX implementations derive from alignment with the
49958 Single UNIX Specification:

- 49959 • The DESCRIPTION states the effect of when *s* is a null pointer.
- 49960 • The [EILSEQ] error condition is added.

49961 The *wcstombs()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

49962 NAME

49963 wcstoul, wcstoull — convert a wide-character string to an unsigned long

49964 SYNOPSIS

49965 #include <wchar.h>

49966 unsigned long wcstoul(const wchar_t *restrict nptr,
49967 wchar_t **restrict endptr, int base);49968 unsigned long long wcstoull(const wchar_t *restrict nptr,
49969 wchar_t **restrict endptr, int base);

49970 DESCRIPTION

49971 cx The functionality described on this reference page is aligned with the ISO C standard. Any
49972 conflict between the requirements described here and the ISO C standard is unintentional. This
49973 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.49974 The *wcstoul()* and *wcstoull()* functions shall convert the initial portion of the wide-character
49975 string pointed to by *nptr* to **unsigned long** and **unsigned long long** representation, respectively.
49976 First, they shall decompose the input wide-character string into three parts:

- 49977 1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by
-
- 49978
- iswspace()*
-)
-
- 49979 2. A subject sequence interpreted as an integer represented in some radix determined by the
-
- 49980 value of
- base*
-
- 49981 3. A final wide-character string of one or more unrecognized wide-character codes, including
-
- 49982 the terminating null wide-character code of the input wide-character string

49983 Then they shall attempt to convert the subject sequence to an unsigned integer, and return the
49984 result.49985 If *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant,
49986 or hexadecimal constant, any of which may be preceded by a '+' or '-' sign. A decimal
49987 constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal
49988 constant consists of the prefix '0' optionally followed by a sequence of the digits '0' to '7'
49989 only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the
49990 decimal digits and letters 'a' (or 'A') to 'f' (or 'F') with values 10 to 15 respectively.49991 If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence
49992 of letters and digits representing an integer with the radix specified by *base*, optionally preceded
49993 by a '+' or '-' sign, but not including an integer suffix. The letters from 'a' (or 'A') to 'z'
49994 (or 'Z') inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less
49995 than that of *base* shall be permitted. If the value of *base* is 16, the wide-character codes 0x or 0X
49996 may optionally precede the sequence of letters and digits, following the sign if present.49997 The subject sequence is defined as the longest initial subsequence of the input wide-character
49998 string, starting with the first wide-character code that is not white space and is of the expected
49999 form. The subject sequence contains no wide-character codes if the input wide-character string is
50000 empty or consists entirely of white-space wide-character codes, or if the first wide-character
50001 code that is not white space is other than a sign or a permissible letter or digit.50002 If the subject sequence has the expected form and *base* is 0, the sequence of wide-character codes
50003 starting with the first digit shall be interpreted as an integer constant. If the subject sequence has
50004 the expected form and the value of *base* is between 2 and 36, it shall be used as the base for
50005 conversion, ascribing to each letter its value as given above. If the subject sequence begins with a
50006 minus sign, the value resulting from the conversion shall be negated. A pointer to the final
50007 wide-character string shall be stored in the object pointed to by *endptr*, provided that *endptr* is

- 50008 not a null pointer.
- 50009 CX In other than the C or POSIX locales, other implementation-defined subject sequences may be
50010 accepted.
- 50011 If the subject sequence is empty or does not have the expected form, no conversion shall be
50012 performed; the value of *nptr* shall be stored in the object pointed to by *endptr*, provided that
50013 *endptr* is not a null pointer.
- 50014 CX The *wcstoul()* function shall not change the setting of *errno* if successful.
- 50015 Since 0, {ULONG_MAX}, and {ULLONG_MAX} are returned on error and 0 is also a valid return
50016 on success, an application wishing to check for error situations should set *errno* to 0, then call
50017 *wcstoul()* or *wcstoull()*, then check *errno*.
- 50018 **RETURN VALUE**
- 50019 Upon successful completion, the *wcstoul()* and *wcstoull()* functions shall return the converted
50020 CX value, if any. If no conversion could be performed, 0 shall be returned and *errno* may be set to
50021 indicate the error. If the correct value is outside the range of representable values,
50022 {ULONG_MAX} or {ULLONG_MAX} respectively shall be returned and *errno* set to [ERANGE].
- 50023 **ERRORS**
- 50024 These functions shall fail if:
- 50025 CX [EINVAL] The value of *base* is not supported.
- 50026 [ERANGE] The value to be returned is not representable.
- 50027 These functions may fail if:
- 50028 CX [EINVAL] No conversion could be performed.
- 50029 **EXAMPLES**
- 50030 None.
- 50031 **APPLICATION USAGE**
- 50032 None.
- 50033 **RATIONALE**
- 50034 None.
- 50035 **FUTURE DIRECTIONS**
- 50036 None.
- 50037 **SEE ALSO**
- 50038 *iswalph()*, *scanf()*, *wcstod()*, *wcstol()*, the Base Definitions volume of IEEE Std 1003.1-2001,
50039 <wchar.h>
- 50040 **CHANGE HISTORY**
- 50041 First released in Issue 4. Derived from the MSE working draft.
- 50042 **Issue 5**
- 50043 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.
- 50044 **Issue 6**
- 50045 Extensions beyond the ISO C standard are marked.
- 50046 The following new requirements on POSIX implementations derive from alignment with the
50047 Single UNIX Specification:
- 50048 • The [EINVAL] error condition is added for when the value of *base* is not supported.

- 50049 In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
50050 added if no conversion could be performed.
- 50051 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:
- 50052 • The *wcstoul()* prototype is updated.
- 50053 • The *wcstoull()* function is added.

50054 **NAME**

50055 wcstoumax — convert a wide-character string to an integer type

50056 **SYNOPSIS**

50057 #include <stddef.h>

50058 #include <inttypes.h>

50059 uintmax_t wcstoumax(const wchar_t *restrict nptr,

50060 wchar_t **restrict endptr, int base);

50061 **DESCRIPTION**

50062 Refer to *wcstoimax()*.

50063 **NAME**50064 wcswcs — find a wide substring (**LEGACY**)50065 **SYNOPSIS**

50066 XSI #include <wchar.h>

50067 wchar_t *wcswcs(const wchar_t *ws1, const wchar_t *ws2);

50068

50069 **DESCRIPTION**

50070 The `wcswcs()` function shall locate the first occurrence in the wide-character string pointed to by
50071 `ws1` of the sequence of wide-character codes (excluding the terminating null wide-character
50072 code) in the wide-character string pointed to by `ws2`.

50073 **RETURN VALUE**

50074 Upon successful completion, `wcswcs()` shall return a pointer to the located wide-character string
50075 or a null pointer if the wide-character string is not found.

50076 If `ws2` points to a wide-character string with zero length, the function shall return `ws1`.

50077 **ERRORS**

50078 No errors are defined.

50079 **EXAMPLES**

50080 None.

50081 **APPLICATION USAGE**

50082 This function was not included in the final ISO/IEC 9899:1990/Amendment 1:1995 (E).
50083 Application developers are strongly encouraged to use the `wcsstr()` function instead.

50084 **RATIONALE**

50085 None.

50086 **FUTURE DIRECTIONS**

50087 This function may be withdrawn in a future version.

50088 **SEE ALSO**50089 `wcschr()`, `wcsstr()`, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>50090 **CHANGE HISTORY**

50091 First released in Issue 4. Derived from the MSE working draft.

50092 **Issue 5**

50093 Marked EX.

50094 **Issue 6**

50095 This function is marked LEGACY.

50096 **NAME**50097 `wcswidth` — number of column positions of a wide-character string50098 **SYNOPSIS**50099 XSI `#include <wchar.h>`50100 `int wcswidth(const wchar_t *pwcs, size_t n);`

50101

50102 **DESCRIPTION**

50103 The `wcswidth()` function shall determine the number of column positions required for *n* wide-character codes (or fewer than *n* wide-character codes if a null wide-character code is encountered before *n* wide-character codes are exhausted) in the string pointed to by *pwcs*.

50106 **RETURN VALUE**

50107 The `wcswidth()` function either shall return 0 (if *pwcs* points to a null wide-character code), or return the number of column positions to be occupied by the wide-character string pointed to by *pwcs*, or return -1 (if any of the first *n* wide-character codes in the wide-character string pointed to by *pwcs* is not a printable wide-character code).

50111 **ERRORS**

50112 No errors are defined.

50113 **EXAMPLES**

50114 None.

50115 **APPLICATION USAGE**

50116 This function was removed from the final ISO/IEC 9899:1990/Amendment 1:1995 (E), and the return value for a non-printable wide character is not specified.

50118 **RATIONALE**

50119 None.

50120 **FUTURE DIRECTIONS**

50121 None.

50122 **SEE ALSO**

50123 `wcwidth()`, the Base Definitions volume of IEEE Std 1003.1-2001, Section 3.103, Column Position,
50124 `<wchar.h>`

50125 **CHANGE HISTORY**

50126 First released in Issue 4. Derived from the MSE working draft.

50127 **Issue 6**

50128 The Open Group Corrigendum U021/11 is applied. The function is marked as an extension.

50129 **NAME**

50130 wcsxfrm — wide-character string transformation

50131 **SYNOPSIS**

50132 #include <wchar.h>

50133 size_t wcsxfrm(wchar_t *restrict ws1, const wchar_t *restrict ws2,
50134 size_t n);

50135 **DESCRIPTION**

50136 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50137 conflict between the requirements described here and the ISO C standard is unintentional. This
50138 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

50139 The *wcsxfrm()* function shall transform the wide-character string pointed to by *ws2* and place the
50140 resulting wide-character string into the array pointed to by *ws1*. The transformation shall be
50141 such that if *wcscmp()* is applied to two transformed wide strings, it shall return a value greater
50142 than, equal to, or less than 0, corresponding to the result of *wcscoll()* applied to the same two
50143 original wide-character strings. No more than *n* wide-character codes shall be placed into the
50144 resulting array pointed to by *ws1*, including the terminating null wide-character code. If *n* is 0,
50145 *ws1* is permitted to be a null pointer. If copying takes place between objects that overlap, the
50146 behavior is undefined.

50147 CX The *wcsxfrm()* function shall not change the setting of *errno* if successful.

50148 Since no return value is reserved to indicate an error, an application wishing to check for error
50149 situations should set *errno* to 0, then call *wcsxfrm()*, then check *errno*.

50150 **RETURN VALUE**

50151 The *wcsxfrm()* function shall return the length of the transformed wide-character string (not
50152 including the terminating null wide-character code). If the value returned is *n* or more, the
50153 contents of the array pointed to by *ws1* are unspecified.

50154 CX On error, the *wcsxfrm()* function may set *errno*, but no return value is reserved to indicate an
50155 error.

50156 **ERRORS**

50157 The *wcsxfrm()* function may fail if:

50158 CX [EINVAL] The wide-character string pointed to by *ws2* contains wide-character codes
50159 outside the domain of the collating sequence.

50160 **EXAMPLES**

50161 None.

50162 **APPLICATION USAGE**

50163 The transformation function is such that two transformed wide-character strings can be ordered
50164 by *wcscmp()* as appropriate to collating sequence information in the program's locale (category
50165 *LC_COLLATE*).

50166 The fact that when *n* is 0 *ws1* is permitted to be a null pointer is useful to determine the size of
50167 the *ws1* array prior to making the transformation.

50168 **RATIONALE**

50169 None.

50170 **FUTURE DIRECTIONS**

50171 None.

50172 **SEE ALSO**50173 `wscmp()`, `wscoll()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<wchar.h>`50174 **CHANGE HISTORY**

50175 First released in Issue 4. Derived from the MSE working draft.

50176 **Issue 5**

50177 Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

50178 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.50179 **Issue 6**50180 In previous versions, this function was required to return `-1` on error.

50181 Extensions beyond the ISO C standard are marked.

50182 The following new requirements on POSIX implementations derive from alignment with the
50183 Single UNIX Specification:

- 50184
- In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
-
- 50185 added if no conversion could be performed.

50186 The `wcsxfrm()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

50187 NAME

50188 wctob — wide-character to single-byte conversion

50189 SYNOPSIS

50190 #include <stdio.h>

50191 #include <wchar.h>

50192 int wctob(wint_t c);

50193 DESCRIPTION

50194 cx The functionality described on this reference page is aligned with the ISO C standard. Any
50195 conflict between the requirements described here and the ISO C standard is unintentional. This
50196 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

50197 The *wctob()* function shall determine whether *c* corresponds to a member of the extended
50198 character set whose character representation is a single byte when in the initial shift state.

50199 The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

50200 RETURN VALUE

50201 The *wctob()* function shall return EOF if *c* does not correspond to a character with length one in
50202 the initial shift state. Otherwise, it shall return the single-byte representation of that character as
50203 an **unsigned char** converted to **int**.

50204 ERRORS

50205 No errors are defined.

50206 EXAMPLES

50207 None.

50208 APPLICATION USAGE

50209 None.

50210 RATIONALE

50211 None.

50212 FUTURE DIRECTIONS

50213 None.

50214 SEE ALSO

50215 *btowc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**wchar.h**>

50216 CHANGE HISTORY

50217 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
50218 (E).

50219 **NAME**

50220 wctomb — convert a wide-character code to a character

50221 **SYNOPSIS**

50222 #include <stdlib.h>

50223 int wctomb(char *s, wchar_t wchar);

50224 **DESCRIPTION**

50225 cx The functionality described on this reference page is aligned with the ISO C standard. Any
 50226 conflict between the requirements described here and the ISO C standard is unintentional. This
 50227 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

50228 The *wctomb()* function shall determine the number of bytes needed to represent the character
 50229 corresponding to the wide-character code whose value is *wchar* (including any change in the
 50230 shift state). It shall store the character representation (possibly multiple bytes and any special
 50231 bytes to change shift state) in the array object pointed to by *s* (if *s* is not a null pointer). At most
 50232 {MB_CUR_MAX} bytes shall be stored. If *wchar* is 0, a null byte shall be stored, preceded by any
 50233 shift sequence needed to restore the initial shift state, and *wctomb()* shall be left in the initial shift
 50234 state.

50235 cx The behavior of this function is affected by the *LC_CTYPE* category of the current locale. For a
 50236 state-dependent encoding, this function shall be placed into its initial state by a call for which its
 50237 character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null
 50238 pointer shall cause the internal state of the function to be altered as necessary. A call with *s* as a
 50239 null pointer shall cause this function to return a non-zero value if encodings have state
 50240 dependency, and 0 otherwise. Changing the *LC_CTYPE* category causes the shift state of this
 50241 function to be unspecified.

50242 The *wctomb()* function need not be reentrant. A function that is not required to be reentrant is
 50243 not required to be thread-safe.

50244 The implementation shall behave as if no function defined in this volume of
 50245 IEEE Std 1003.1-2001 calls *wctomb()*.

50246 **RETURN VALUE**

50247 If *s* is a null pointer, *wctomb()* shall return a non-zero or 0 value, if character encodings,
 50248 respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, *wctomb()*
 50249 shall return -1 if the value of *wchar* does not correspond to a valid character, or return the
 50250 number of bytes that constitute the character corresponding to the value of *wchar*.

50251 In no case shall the value returned be greater than the value of the {MB_CUR_MAX} macro.

50252 **ERRORS**

50253 No errors are defined.

50254 **EXAMPLES**

50255 None.

50256 **APPLICATION USAGE**

50257 None.

50258 **RATIONALE**

50259 None.

50260 **FUTURE DIRECTIONS**

50261 None.

50262 **SEE ALSO**

50263 *mblen()*, *mbtowc()*, *mbstowcs()*, *wcstombs()*, the Base Definitions volume of IEEE Std 1003.1-2001,
50264 **<stdlib.h>**

50265 **CHANGE HISTORY**

50266 First released in Issue 4. Derived from the ANSI C standard.

50267 **Issue 6**

50268 Extensions beyond the ISO C standard are marked.

50269 In the DESCRIPTION, a note about reentrancy and thread-safety is added.

50270 **NAME**

50271 wctrans — define character mapping

50272 **SYNOPSIS**

50273 #include <wctype.h>

50274 wctrans_t wctrans(const char *charclass);

50275 **DESCRIPTION**

50276 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 50277 conflict between the requirements described here and the ISO C standard is unintentional. This
 50278 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

50279 The *wctrans()* function is defined for valid character mapping names identified in the current
 50280 locale. The *charclass* is a string identifying a generic character mapping name for which codeset-
 50281 specific information is required. The following character mapping names are defined in all
 50282 locales: **tolower** and **toupper**.

50283 The function shall return a value of type **wctrans_t**, which can be used as the second argument
 50284 to subsequent calls of *towctrans()*. The *wctrans()* function shall determine values of **wctrans_t**
 50285 according to the rules of the coded character set defined by character mapping information in
 50286 the program's locale (category *LC_CTYPE*). The values returned by *wctrans()* shall be valid until
 50287 a call to *setlocale()* that modifies the category *LC_CTYPE*.

50288 **RETURN VALUE**

50289 CX The *wctrans()* function shall return 0 and may set *errno* to indicate the error if the given
 50290 character mapping name is not valid for the current locale (category *LC_CTYPE*); otherwise, it
 50291 shall return a non-zero object of type **wctrans_t** that can be used in calls to *towctrans()*.

50292 **ERRORS**50293 The *wctrans()* function may fail if:

50294 CX [EINVAL] The character mapping name pointed to by *charclass* is not valid in the current
 50295 locale.

50296 **EXAMPLES**

50297 None.

50298 **APPLICATION USAGE**

50299 None.

50300 **RATIONALE**

50301 None.

50302 **FUTURE DIRECTIONS**

50303 None.

50304 **SEE ALSO**50305 *towctrans()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wctype.h>50306 **CHANGE HISTORY**

50307 First released in Issue 5. Derived from ISO/IEC 9899:1990/Amendment 1:1995 (E).

50308 **NAME**

50309 wctype — define character class

50310 **SYNOPSIS**

50311 #include <wctype.h>

50312 wctype_t wctype(const char *property);

50313 **DESCRIPTION**

50314 cx The functionality described on this reference page is aligned with the ISO C standard. Any
 50315 conflict between the requirements described here and the ISO C standard is unintentional. This
 50316 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

50317 The *wctype()* function is defined for valid character class names as defined in the current locale.
 50318 The *property* argument is a string identifying a generic character class for which codeset-specific
 50319 type information is required. The following character class names shall be defined in all locales:

50320	alnum	digit	punct
50321	alpha	graph	space
50322	blank	lower	upper
50323	cntrl	print	xdigit

50324 Additional character class names defined in the locale definition file (category *LC_CTYPE*) can
 50325 also be specified.

50326 The function shall return a value of type **wctype_t**, which can be used as the second argument to
 50327 subsequent calls of *iswctype()*. The *wctype()* function shall determine values of **wctype_t**
 50328 according to the rules of the coded character set defined by character type information in the
 50329 program's locale (category *LC_CTYPE*). The values returned by *wctype()* shall be valid until a
 50330 call to *setlocale()* that modifies the category *LC_CTYPE*.

50331 **RETURN VALUE**

50332 The *wctype()* function shall return 0 if the given character class name is not valid for the current
 50333 locale (category *LC_CTYPE*); otherwise, it shall return an object of type **wctype_t** that can be
 50334 used in calls to *iswctype()*.

50335 **ERRORS**

50336 No errors are defined.

50337 **EXAMPLES**

50338 None.

50339 **APPLICATION USAGE**

50340 None.

50341 **RATIONALE**

50342 None.

50343 **FUTURE DIRECTIONS**

50344 None.

50345 **SEE ALSO**50346 *iswctype()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**wctype.h**>50347 **CHANGE HISTORY**

50348 First released in Issue 4.

50349 **Issue 5**

50350 The following change has been made in this issue for alignment with
50351 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 50352 • The SYNOPSIS has been changed to indicate that this function and associated data types are
50353 now made visible by inclusion of the `<wctype.h>` header rather than `<wchar.h>`.

50354 NAME

50355 wcwidth — number of column positions of a wide-character code

50356 SYNOPSIS

50357 xSI #include <wchar.h>

50358 int wcwidth(wchar_t wc);

50359

50360 DESCRIPTION

50361 The *wcwidth()* function shall determine the number of column positions required for the wide
50362 character *wc*. The application shall ensure that the value of *wc* is a character representable as a
50363 **wchar_t**, and is a wide-character code corresponding to a valid character in the current locale.

50364 RETURN VALUE

50365 The *wcwidth()* function shall either return 0 (if *wc* is a null wide-character code), or return the
50366 number of column positions to be occupied by the wide-character code *wc*, or return *-1* (if *wc*
50367 does not correspond to a printable wide-character code).

50368 ERRORS

50369 No errors are defined.

50370 EXAMPLES

50371 None.

50372 APPLICATION USAGE

50373 This function was removed from the final ISO/IEC 9899:1990/Amendment 1:1995 (E), and the
50374 return value for a non-printable wide character is not specified.

50375 RATIONALE

50376 None.

50377 FUTURE DIRECTIONS

50378 None.

50379 SEE ALSO

50380 *wcswidth()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**wchar.h**>

50381 CHANGE HISTORY

50382 First released as a World-wide Portability Interface in Issue 4. Derived from the MSE working
50383 draft.

50384 Issue 6

50385 The Open Group Corrigendum U021/12 is applied. This function is marked as an extension.

50386 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

50387 **NAME**

50388 wmemchr — find a wide character in memory

50389 **SYNOPSIS**

50390 #include <wchar.h>

50391 wchar_t *wmemchr(const wchar_t *ws, wchar_t wc, size_t n);

50392 **DESCRIPTION**

50393 cx The functionality described on this reference page is aligned with the ISO C standard. Any
50394 conflict between the requirements described here and the ISO C standard is unintentional. This
50395 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

50396 The *wmemchr()* function shall locate the first occurrence of *wc* in the initial *n* wide characters of
50397 the object pointed to by *ws*. This function shall not be affected by locale and all **wchar_t** values
50398 shall be treated identically. The null wide character and **wchar_t** values not corresponding to
50399 valid characters shall not be treated specially.

50400 If *n* is zero, the application shall ensure that *ws* is a valid pointer and the function behaves as if
50401 no valid occurrence of *wc* is found.

50402 **RETURN VALUE**

50403 The *wmemchr()* function shall return a pointer to the located wide character, or a null pointer if
50404 the wide character does not occur in the object.

50405 **ERRORS**

50406 No errors are defined.

50407 **EXAMPLES**

50408 None.

50409 **APPLICATION USAGE**

50410 None.

50411 **RATIONALE**

50412 None.

50413 **FUTURE DIRECTIONS**

50414 None.

50415 **SEE ALSO**

50416 *wmemcmp()*, *wmemcpy()*, *wmemmove()*, *wmemset()*, the Base Definitions volume of
50417 IEEE Std 1003.1-2001, <**wchar.h**>

50418 **CHANGE HISTORY**

50419 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
50420 (E).

50421 **Issue 6**

50422 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

50423 **NAME**

50424 wmemcmp — compare wide characters in memory

50425 **SYNOPSIS**

50426 #include <wchar.h>

50427 int wmemcmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);

50428 **DESCRIPTION**

50429 cx The functionality described on this reference page is aligned with the ISO C standard. Any
50430 conflict between the requirements described here and the ISO C standard is unintentional. This
50431 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

50432 The *wmemcmp()* function shall compare the first *n* wide characters of the object pointed to by
50433 *ws1* to the first *n* wide characters of the object pointed to by *ws2*. This function shall not be
50434 affected by locale and all **wchar_t** values shall be treated identically. The null wide character and
50435 **wchar_t** values not corresponding to valid characters shall not be treated specially.

50436 If *n* is zero, the application shall ensure that *ws1* and *ws2* are valid pointers, and the function
50437 shall behave as if the two objects compare equal.

50438 **RETURN VALUE**

50439 The *wmemcmp()* function shall return an integer greater than, equal to, or less than zero,
50440 respectively, as the object pointed to by *ws1* is greater than, equal to, or less than the object
50441 pointed to by *ws2*.

50442 **ERRORS**

50443 No errors are defined.

50444 **EXAMPLES**

50445 None.

50446 **APPLICATION USAGE**

50447 None.

50448 **RATIONALE**

50449 None.

50450 **FUTURE DIRECTIONS**

50451 None.

50452 **SEE ALSO**

50453 *wmemchr()*, *wmemcpy()*, *wmemmove()*, *wmemset()*, the Base Definitions volume of
50454 IEEE Std 1003.1-2001, <wchar.h>

50455 **CHANGE HISTORY**

50456 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
50457 (E).

50458 **Issue 6**

50459 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

50460 **NAME**

50461 wmemcpy — copy wide characters in memory

50462 **SYNOPSIS**

50463 #include <wchar.h>

50464 wchar_t *wmemcpy(wchar_t *restrict ws1, const wchar_t *restrict ws2,
50465 size_t n);50466 **DESCRIPTION**50467 cx The functionality described on this reference page is aligned with the ISO C standard. Any
50468 conflict between the requirements described here and the ISO C standard is unintentional. This
50469 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.50470 The *wmemcpy()* function shall copy *n* wide characters from the object pointed to by *ws2* to the
50471 object pointed to by *ws1*. This function shall not be affected by locale and all **wchar_t** values
50472 shall be treated identically. The null wide character and **wchar_t** values not corresponding to
50473 valid characters shall not be treated specially.50474 If *n* is zero, the application shall ensure that *ws1* and *ws2* are valid pointers, and the function
50475 shall copy zero wide characters.50476 **RETURN VALUE**50477 The *wmemcpy()* function shall return the value of *ws1*.50478 **ERRORS**

50479 No errors are defined.

50480 **EXAMPLES**

50481 None.

50482 **APPLICATION USAGE**

50483 None.

50484 **RATIONALE**

50485 None.

50486 **FUTURE DIRECTIONS**

50487 None.

50488 **SEE ALSO**50489 *wmemchr()*, *wmemcmp()*, *wmemmove()*, *wmemset()*, the Base Definitions volume of
50490 IEEE Std 1003.1-2001, <**wchar.h**>50491 **CHANGE HISTORY**50492 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
50493 (E).50494 **Issue 6**

50495 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

50496 The *wmemcpy()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

50497 **NAME**

50498 wmemmove — copy wide characters in memory with overlapping areas

50499 **SYNOPSIS**

50500 #include <wchar.h>

50501 wchar_t *wmemmove(wchar_t *ws1, const wchar_t *ws2, size_t n);

50502 **DESCRIPTION**

50503 cx The functionality described on this reference page is aligned with the ISO C standard. Any
 50504 conflict between the requirements described here and the ISO C standard is unintentional. This
 50505 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

50506 The *wmemmove()* function shall copy *n* wide characters from the object pointed to by *ws2* to the
 50507 object pointed to by *ws1*. Copying shall take place as if the *n* wide characters from the object
 50508 pointed to by *ws2* are first copied into a temporary array of *n* wide characters that does not
 50509 overlap the objects pointed to by *ws1* or *ws2*, and then the *n* wide characters from the temporary
 50510 array are copied into the object pointed to by *ws1*.

50511 This function shall not be affected by locale and all **wchar_t** values shall be treated identically.
 50512 The null wide character and **wchar_t** values not corresponding to valid characters shall not be
 50513 treated specially.

50514 If *n* is zero, the application shall ensure that *ws1* and *ws2* are valid pointers, and the function
 50515 shall copy zero wide characters.

50516 **RETURN VALUE**50517 The *wmemmove()* function shall return the value of *ws1*.50518 **ERRORS**

50519 No errors are defined

50520 **EXAMPLES**

50521 None.

50522 **APPLICATION USAGE**

50523 None.

50524 **RATIONALE**

50525 None.

50526 **FUTURE DIRECTIONS**

50527 None.

50528 **SEE ALSO**

50529 *wmemchr()*, *wmemcmp()*, *wmemcpy()*, *wmemset()*, the Base Definitions volume of
 50530 IEEE Std 1003.1-2001, <wchar.h>

50531 **CHANGE HISTORY**

50532 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
 50533 (E).

50534 **Issue 6**

50535 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

50536 **NAME**

50537 wmemset — set wide characters in memory

50538 **SYNOPSIS**

50539 #include <wchar.h>

50540 wchar_t *wmemset(wchar_t *ws, wchar_t wc, size_t n);

50541 **DESCRIPTION**

50542 cx The functionality described on this reference page is aligned with the ISO C standard. Any
50543 conflict between the requirements described here and the ISO C standard is unintentional. This
50544 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

50545 The *wmemset()* function shall copy the value of *wc* into each of the first *n* wide characters of the
50546 object pointed to by *ws*. This function shall not be affected by locale and all **wchar_t** values shall
50547 be treated identically. The null wide character and **wchar_t** values not corresponding to valid
50548 characters shall not be treated specially.

50549 If *n* is zero, the application shall ensure that *ws* is a valid pointer, and the function shall copy
50550 zero wide characters.

50551 **RETURN VALUE**50552 The *wmemset()* functions shall return the value of *ws*.50553 **ERRORS**

50554 No errors are defined.

50555 **EXAMPLES**

50556 None.

50557 **APPLICATION USAGE**

50558 None.

50559 **RATIONALE**

50560 None.

50561 **FUTURE DIRECTIONS**

50562 None.

50563 **SEE ALSO**

50564 *wmemchr()*, *wmemcmp()*, *wmemcpy()*, *wmemmove()*, the Base Definitions volume of
50565 IEEE Std 1003.1-2001, <**wchar.h**>

50566 **CHANGE HISTORY**

50567 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
50568 (E).

50569 **Issue 6**

50570 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

50571 NAME

50572 wordexp, wordfree — perform word expansions

50573 SYNOPSIS

50574 #include <wordexp.h>

50575 int wordexp(const char *restrict words, wordexp_t *restrict pwordexp,
50576 int flags);

50577 void wordfree(wordexp_t *pwordexp);

50578 DESCRIPTION

50579 The *wordexp()* function shall perform word expansions as described in the Shell and Utilities
 50580 volume of IEEE Std 1003.1-2001, Section 2.6, Word Expansions, subject to quoting as in the Shell
 50581 and Utilities volume of IEEE Std 1003.1-2001, Section 2.2, Quoting, and place the list of expanded
 50582 words into the structure pointed to by *pwordexp*.

50583 The *words* argument is a pointer to a string containing one or more words to be expanded. The
 50584 expansions shall be the same as would be performed by the command line interpreter if *words*
 50585 were the part of a command line representing the arguments to a utility. Therefore, the
 50586 application shall ensure that *words* does not contain an unquoted <newline> or any of the
 50587 unquoted shell special characters *'|', '&', ';', '<', '>'* except in the context of command
 50588 substitution as specified in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.6.3,
 50589 Command Substitution. It also shall not contain unquoted parentheses or braces, except in the
 50590 context of command or variable substitution. The application shall ensure that every member of
 50591 *words* which it expects to have expanded by *wordexp()* does not contain an unquoted initial
 50592 comment character. The application shall also ensure that any words which it intends to be
 50593 ignored (because they begin or continue a comment) are deleted from *words*. If the argument
 50594 *words* contains an unquoted comment character (number sign) that is the beginning of a token,
 50595 *wordexp()* shall either treat the comment character as a regular character, or interpret it as a
 50596 comment indicator and ignore the remainder of *words*.

50597 The structure type **wordexp_t** is defined in the **<wordexp.h>** header and includes at least the
 50598 following members:

50599

50600

Member Type	Member Name	Description
size_t	<i>we_wordc</i>	Count of words matched by <i>words</i> .
char **	<i>we_wordv</i>	Pointer to list of expanded words.
size_t	<i>we_offs</i>	Slots to reserve at the beginning of <i>pwordexp->we_wordv</i> .

50603

50604 The *wordexp()* function shall store the number of generated words into *pwordexp->we_wordc* and
 50605 a pointer to a list of pointers to words in *pwordexp->we_wordv*. Each individual field created
 50606 during field splitting (see the Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.6.5,
 50607 Field Splitting) or pathname expansion (see the Shell and Utilities volume of
 50608 IEEE Std 1003.1-2001, Section 2.6.6, Pathname Expansion) shall be a separate word in the
 50609 *pwordexp->we_wordv* list. The words shall be in order as described in the Shell and Utilities
 50610 volume of IEEE Std 1003.1-2001, Section 2.6, Word Expansions. The first pointer after the last
 50611 word pointer shall be a null pointer. The expansion of special parameters described in the Shell
 50612 and Utilities volume of IEEE Std 1003.1-2001, Section 2.5.2, Special Parameters is unspecified.

50613 It is the caller's responsibility to allocate the storage pointed to by *pwordexp*. The *wordexp()*
 50614 function shall allocate other space as needed, including memory pointed to by
 50615 *pwordexp->we_wordv*. The *wordfree()* function frees any memory associated with *pwordexp* from a
 50616 previous call to *wordexp()*.

50617 The *flags* argument is used to control the behavior of *wordexp()*. The value of *flags* is the
 50618 bitwise-inclusive OR of zero or more of the following constants, which are defined in
 50619 **<wordexp.h>**:

50620 **WRDE_APPEND** Append words generated to the ones from a previous call to *wordexp()*.

50621 **WRDE_DOOFFS** Make use of *pwordexp->we_offs*. If this flag is set, *pwordexp->we_offs* is used
 50622 to specify how many null pointers to add to the beginning of
 50623 *pwordexp->we_wordv*. In other words, *pwordexp->we_wordv* shall point to
 50624 *pwordexp->we_offs* null pointers, followed by *pwordexp->we_wordc* word
 50625 pointers, followed by a null pointer.

50626 **WRDE_NOCMD** If the implementation supports the utilities defined in the Shell and
 50627 Utilities volume of IEEE Std 1003.1-2001, fail if command substitution, as
 50628 specified in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section
 50629 2.6.3, Command Substitution, is requested.

50630 **WRDE_REUSE** The *pwordexp* argument was passed to a previous successful call to
 50631 *wordexp()*, and has not been passed to *wordfree()*. The result shall be the
 50632 same as if the application had called *wordfree()* and then called *wordexp()*
 50633 without **WRDE_REUSE**.

50634 **WRDE_SHOWERR** Do not redirect *stderr* to **/dev/null**.

50635 **WRDE_UNDEF** Report error on an attempt to expand an undefined shell variable.

50636 The **WRDE_APPEND** flag can be used to append a new set of words to those generated by a
 50637 previous call to *wordexp()*. The following rules apply to applications when two or more calls to
 50638 *wordexp()* are made with the same value of *pwordexp* and without intervening calls to *wordfree()*:

- 50639 1. The first such call shall not set **WRDE_APPEND**. All subsequent calls shall set it.
- 50640 2. All of the calls shall set **WRDE_DOOFFS**, or all shall not set it.
- 50641 3. After the second and each subsequent call, *pwordexp->we_wordv* shall point to a list
 50642 containing the following:
 - 50643 a. Zero or more null pointers, as specified by **WRDE_DOOFFS** and *pwordexp->we_offs*
 - 50644 b. Pointers to the words that were in the *pwordexp->we_wordv* list before the call, in the
 50645 same order as before
 - 50646 c. Pointers to the new words generated by the latest call, in the specified order
- 50647 4. The count returned in *pwordexp->we_wordc* shall be the total number of words from all of
 50648 the calls.
- 50649 5. The application can change any of the fields after a call to *wordexp()*, but if it does it shall
 50650 reset them to the original value before a subsequent call, using the same *pwordexp* value, to
 50651 *wordfree()* or *wordexp()* with the **WRDE_APPEND** or **WRDE_REUSE** flag.

50652 If the implementation supports the utilities defined in the Shell and Utilities volume of
 50653 IEEE Std 1003.1-2001, and *words* contains an unquoted character—<newline>, ' | ', ' & ', ' ; ',
 50654 ' < ', ' > ', ' (', ') ', ' { ', ' } '—in an inappropriate context, *wordexp()* shall fail, and the number
 50655 of expanded words shall be 0.

50656 Unless **WRDE_SHOWERR** is set in *flags*, *wordexp()* shall redirect *stderr* to **/dev/null** for any
 50657 utilities executed as a result of command substitution while expanding *words*. If
 50658 **WRDE_SHOWERR** is set, *wordexp()* may write messages to *stderr* if syntax errors are detected
 50659 while expanding *words*.

50660 The application shall ensure that if WRDE_DOOFFS is set, then *pwordexp->we_offs* has the same
 50661 value for each *wordexp()* call and *wordfree()* call using a given *pwordexp*.

50662 The following constants are defined as error return values:

50663 50664	WRDE_BADCHAR	One of the unquoted characters—<newline>, ' ', '&', ';', '<', '>', '(', ')', '{', '}'—appears in <i>words</i> in an inappropriate context.
50665	WRDE_BADVAL	Reference to undefined shell variable when WRDE_UNDEF is set in <i>flags</i> .
50666	WRDE_CMDSUB	Command substitution requested when WRDE_NOCMD was set in <i>flags</i> .
50667	WRDE_NOSPACE	Attempt to allocate memory failed.
50668 50669	WRDE_SYNTAX	Shell syntax error, such as unbalanced parentheses or unterminated string.

50670 RETURN VALUE

50671 Upon successful completion, *wordexp()* shall return 0. Otherwise, a non-zero value, as described
 50672 in <**wordexp.h**>, shall be returned to indicate an error. If *wordexp()* returns the value
 50673 WRDE_NOSPACE, then *pwordexp->we_wordc* and *pwordexp->we_wordv* shall be updated to
 50674 reflect any words that were successfully expanded. In other cases, they shall not be modified.

50675 The *wordfree()* function shall not return a value.

50676 ERRORS

50677 No errors are defined.

50678 EXAMPLES

50679 None.

50680 APPLICATION USAGE

50681 The *wordexp()* function is intended to be used by an application that wants to do all of the shell's
 50682 expansions on a word or words obtained from a user. For example, if the application prompts
 50683 for a filename (or list of filenames) and then uses *wordexp()* to process the input, the user could
 50684 respond with anything that would be valid as input to the shell.

50685 The WRDE_NOCMD flag is provided for applications that, for security or other reasons, want to
 50686 prevent a user from executing shell commands. Disallowing unquoted shell special characters
 50687 also prevents unwanted side effects, such as executing a command or writing a file.

50688 RATIONALE

50689 This function was included as an alternative to *glob()*. There had been continuing controversy
 50690 over exactly what features should be included in *glob()*. It is hoped that by providing *wordexp()*
 50691 (which provides all of the shell word expansions, but which may be slow to execute) and *glob()*
 50692 (which is faster, but which only performs pathname expansion, without tilde or parameter
 50693 expansion) this will satisfy the majority of applications.

50694 While *wordexp()* could be implemented entirely as a library routine, it is expected that most
 50695 implementations run a shell in a subprocess to do the expansion.

50696 Two different approaches have been proposed for how the required information might be
 50697 presented to the shell and the results returned. They are presented here as examples.

50698 One proposal is to extend the *echo* utility by adding a **-q** option. This option would cause *echo* to
 50699 add a backslash before each backslash and <blank> that occurs within an argument. The
 50700 *wordexp()* function could then invoke the shell as follows:

```
50701 (void) strcpy(buffer, "echo -q");
50702 (void) strcat(buffer, words);
50703 if ((flags & WRDE_SHOWERR) == 0)
```



```

50704         (void) strcat(buffer, "2>/dev/null");
50705     f = popen(buffer, "r");

```

50706 The *wordexp()* function would read the resulting output, remove unquoted backslashes, and
 50707 break into words at unquoted <blank>s. If the WRDE_NOCMD flag was set, *wordexp()* would
 50708 have to scan *words* before starting the subshell to make sure that there would be no command
 50709 substitution. In any case, it would have to scan *words* for unquoted special characters.

50710 Another proposal is to add the following options to *sh*:

50711 **-w wordlist**

50712 This option provides a wordlist expansion service to applications. The words in *wordlist*
 50713 shall be expanded and the following written to standard output:

- 50714 1. The count of the number of words after expansion, in decimal, followed by a null byte
- 50715 2. The number of bytes needed to represent the expanded words (not including null
 50716 separators), in decimal, followed by a null byte
- 50717 3. The expanded words, each terminated by a null byte

50718 If an error is encountered during word expansion, *sh* exits with a non-zero status after
 50719 writing the former to report any words successfully expanded

50720 **-P** Run in “protected” mode. If specified with the **-w** option, no command substitution shall
 50721 be performed.

50722 With these options, *wordexp()* could be implemented fairly simply by creating a subprocess
 50723 using *fork()* and executing *sh* using the line:

```

50724 execl(<shell path>, "sh", "-P", "-w", words, (char *)0);

```

50725 after directing standard error to **/dev/null**.

50726 It seemed objectionable for a library routine to write messages to standard error, unless
 50727 explicitly requested, so *wordexp()* is required to redirect standard error to **/dev/null** to ensure
 50728 that no messages are generated, even for commands executed for command substitution. The
 50729 WRDE_SHOWERR flag can be specified to request that error messages be written.

50730 The WRDE_REUSE flag allows the implementation to avoid the expense of freeing and
 50731 reallocating memory, if that is possible. A minimal implementation can call *wordfree()* when
 50732 WRDE_REUSE is set.

50733 FUTURE DIRECTIONS

50734 None.

50735 SEE ALSO

50736 *fnmatch()*, *glob()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<wordexp.h>**, the Shell
 50737 and Utilities volume of IEEE Std 1003.1-2001, Chapter 2, Shell Command Language

50738 CHANGE HISTORY

50739 First released in Issue 4. Derived from the ISO POSIX-2 standard.

50740 Issue 5

50741 Moved from POSIX2 C-language Binding to BASE.

50742 Issue 6

50743 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

50744 The **restrict** keyword is added to the *wordexp()* prototype for alignment with the
 50745 ISO/IEC 9899:1999 standard.

50746 **NAME**50747 **wprintf** — print formatted wide-character output50748 **SYNOPSIS**50749 `#include <stdio.h>`50750 `#include <wchar.h>`50751 `int wprintf(const wchar_t *restrict format, ...);`50752 **DESCRIPTION**50753 Refer to *fwprintf()*.

50754 **NAME**

50755 pwrite, write — write on a file

50756 **SYNOPSIS**

50757 #include <unistd.h>

```
50758 xsi ssize_t pwrite(int fildes, const void *buf, size_t nbyte,
50759                off_t offset);
```

```
50760 ssize_t write(int fildes, const void *buf, size_t nbyte);
```

50761 **DESCRIPTION**

50762 The *write()* function shall attempt to write *nbyte* bytes from the buffer pointed to by *buf* to the
 50763 file associated with the open file descriptor, *fildes*.

50764 Before any action described below is taken, and if *nbyte* is zero and the file is a regular file, the
 50765 *write()* function may detect and return errors as described below. In the absence of errors, or if
 50766 error detection is not performed, the *write()* function shall return zero and have no other results.
 50767 If *nbyte* is zero and the file is not a regular file, the results are unspecified.

50768 On a regular file or other file capable of seeking, the actual writing of data shall proceed from the
 50769 position in the file indicated by the file offset associated with *fildes*. Before successful return
 50770 from *write()*, the file offset shall be incremented by the number of bytes actually written. On a
 50771 regular file, if this incremented file offset is greater than the length of the file, the length of the
 50772 file shall be set to this file offset.

50773 On a file not capable of seeking, writing shall always take place starting at the current position.
 50774 The value of a file offset associated with such a device is undefined.

50775 If the O_APPEND flag of the file status flags is set, the file offset shall be set to the end of the file
 50776 prior to each write and no intervening file modification operation shall occur between changing
 50777 the file offset and the write operation.

50778 xsi If a *write()* requests that more bytes be written than there is room for (for example, the process'
 50779 file size limit or the physical end of a medium), only as many bytes as there is room for shall be
 50780 written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A
 50781 write of 512 bytes will return 20. The next write of a non-zero number of bytes would give a
 50782 failure return (except as noted below).

50783 xsi If the request would cause the file size to exceed the soft file size limit for the process and there
 50784 is no room for any bytes to be written, the request shall fail and the implementation shall
 50785 generate the SIGXFSZ signal for the thread.

50786 If *write()* is interrupted by a signal before it writes any data, it shall return -1 with *errno* set to
 50787 [EINTR].

50788 If *write()* is interrupted by a signal after it successfully writes some data, it shall return the
 50789 number of bytes written.

50790 If the value of *nbyte* is greater than {SSIZE_MAX}, the result is implementation-defined.

50791 After a *write()* to a regular file has successfully returned:

- 50792 • Any successful *read()* from each byte position in the file that was modified by that write shall
- 50793 return the data specified by the *write()* for that position until such byte positions are again
- 50794 modified.
- 50795 • Any subsequent successful *write()* to the same byte position in the file shall overwrite that
- 50796 file data.

50797	Write requests to a pipe or FIFO shall be handled in the same way as a regular file with the
50798	following exceptions:
50799	• There is no file offset associated with a pipe, hence each write request shall append to the
50800	end of the pipe.
50801	• Write requests of {PIPE_BUF} bytes or less shall not be interleaved with data from other
50802	processes doing writes on the same pipe. Writes of greater than {PIPE_BUF} bytes may have
50803	data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the
50804	O_NONBLOCK flag of the file status flags is set.
50805	• If the O_NONBLOCK flag is clear, a write request may cause the thread to block, but on
50806	normal completion it shall return <i>nbyte</i> .
50807	• If the O_NONBLOCK flag is set, <i>write()</i> requests shall be handled differently, in the
50808	following ways:
50809	— The <i>write()</i> function shall not block the thread.
50810	— A write request for {PIPE_BUF} or fewer bytes shall have the following effect: if there is
50811	sufficient space available in the pipe, <i>write()</i> shall transfer all the data and return the
50812	number of bytes requested. Otherwise, <i>write()</i> shall transfer no data and return -1 with
50813	<i>errno</i> set to [EAGAIN].
50814	— A write request for more than {PIPE_BUF} bytes shall cause one of the following:
50815	— When at least one byte can be written, transfer what it can and return the number of
50816	bytes written. When all data previously written to the pipe is read, it shall transfer at
50817	least {PIPE_BUF} bytes.
50818	— When no data can be written, transfer no data, and return -1 with <i>errno</i> set to
50819	[EAGAIN].
50820	When attempting to write to a file descriptor (other than a pipe or FIFO) that supports non-
50821	blocking writes and cannot accept the data immediately:
50822	• If the O_NONBLOCK flag is clear, <i>write()</i> shall block the calling thread until the data can be
50823	accepted.
50824	• If the O_NONBLOCK flag is set, <i>write()</i> shall not block the thread. If some data can be
50825	written without blocking the thread, <i>write()</i> shall write what it can and return the number of
50826	bytes written. Otherwise, it shall return -1 and set <i>errno</i> to [EAGAIN].
50827	Upon successful completion, where <i>nbyte</i> is greater than 0, <i>write()</i> shall mark for update the
50828	<i>st_ctime</i> and <i>st_mtime</i> fields of the file, and if the file is a regular file, the S_ISUID and S_ISGID
50829	bits of the file mode may be cleared.
50830	For regular files, no data transfer shall occur past the offset maximum established in the open
50831	file description associated with <i>fildes</i> .
50832	If <i>fildes</i> refers to a socket, <i>write()</i> shall be equivalent to <i>send()</i> with no flags set.
50833 SIO	If the O_DSYNC bit has been set, write I/O operations on the file descriptor shall complete as
50834	defined by synchronized I/O data integrity completion.
50835	If the O_SYNC bit has been set, write I/O operations on the file descriptor shall complete as
50836	defined by synchronized I/O file integrity completion.
50837 SHM	If <i>fildes</i> refers to a shared memory object, the result of the <i>write()</i> function is unspecified.
50838 TYM	If <i>fildes</i> refers to a typed memory object, the result of the <i>write()</i> function is unspecified.

50839 XSR If *fildev* refers to a STREAM, the operation of *write()* shall be determined by the values of the
 50840 minimum and maximum *nbyte* range (packet size) accepted by the STREAM. These values are
 50841 determined by the topmost STREAM module. If *nbyte* falls within the packet size range, *nbyte*
 50842 bytes shall be written. If *nbyte* does not fall within the range and the minimum packet size value
 50843 is 0, *write()* shall break the buffer into maximum packet size segments prior to sending the data
 50844 downstream (the last segment may contain less than the maximum packet size). If *nbyte* does not
 50845 fall within the range and the minimum value is non-zero, *write()* shall fail with *errno* set to
 50846 [ERANGE]. Writing a zero-length buffer (*nbyte* is 0) to a STREAMS device sends 0 bytes with 0
 50847 returned. However, writing a zero-length buffer to a STREAMS-based pipe or FIFO sends no
 50848 message and 0 is returned. The process may issue *I_SWROPT ioctl()* to enable zero-length
 50849 messages to be sent across the pipe or FIFO.

50850 When writing to a STREAM, data messages are created with a priority band of 0. When writing
 50851 to a STREAM that is not a pipe or FIFO:

50852 • If O_NONBLOCK is clear, and the STREAM cannot accept data (the STREAM write queue is
 50853 full due to internal flow control conditions), *write()* shall block until data can be accepted.

50854 • If O_NONBLOCK is set and the STREAM cannot accept data, *write()* shall return -1 and set
 50855 *errno* to [EAGAIN].

50856 • If O_NONBLOCK is set and part of the buffer has been written while a condition in which
 50857 the STREAM cannot accept additional data occurs, *write()* shall terminate and return the
 50858 number of bytes written.

50859 In addition, *write()* shall fail if the STREAM head has processed an asynchronous error before
 50860 the call. In this case, the value of *errno* does not reflect the result of *write()*, but reflects the prior
 50861 error.

50862 XSI The *pwrite()* function shall be equivalent to *write()*, except that it writes into a given position
 50863 without changing the file pointer. The first three arguments to *pwrite()* are the same as *write()*
 50864 with the addition of a fourth argument offset for the desired position inside the file.

50865 RETURN VALUE

50866 XSI Upon successful completion, *write()* and *pwrite()* shall return the number of bytes actually
 50867 written to the file associated with *fildev*. This number shall never be greater than *nbyte*.
 50868 Otherwise, -1 shall be returned and *errno* set to indicate the error.

50869 ERRORS

50870 XSI The *write()* and *pwrite()* functions shall fail if:

50871 [EAGAIN] The O_NONBLOCK flag is set for the file descriptor and the thread would be
 50872 delayed in the *write()* operation.

50873 [EBADF] The *fildev* argument is not a valid file descriptor open for writing.

50874 [EFBIG] An attempt was made to write a file that exceeds the implementation-defined
 50875 XSI maximum file size or the process' file size limit, and there was no room for
 50876 any bytes to be written.

50877 [EFBIG] The file is a regular file, *nbyte* is greater than 0, and the starting position is
 50878 greater than or equal to the offset maximum established in the open file
 50879 description associated with *fildev*.

50880 [EINTR] The write operation was terminated due to the receipt of a signal, and no data
 50881 was transferred.

50882 [EIO] The process is a member of a background process group attempting to write
 50883 to its controlling terminal, TOSTOP is set, the process is neither ignoring nor

50884		blocking SIGTTOU, and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions.
50885		
50886	[ENOSPC]	There was no free space remaining on the device containing the file.
50887	[EPIPE]	An attempt is made to write to a pipe or FIFO that is not open for reading by any process, or that only has one end open. A SIGPIPE signal shall also be sent to the thread.
50888		
50889		
50890 XSR	[ERANGE]	The transfer request size was outside the range supported by the STREAMS file associated with <i>fildev</i> .
50891		
50892		The <i>write()</i> function shall fail if:
50893	[EAGAIN] or [EWOULDBLOCK]	
50894		The file descriptor is for a socket, is marked O_NONBLOCK, and write would block.
50895		
50896	[ECONNRESET]	A write was attempted on a socket that is not connected.
50897	[EPIPE]	A write was attempted on a socket that is shut down for writing, or is no longer connected. In the latter case, if the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling process.
50898		
50899		
50900 XSI		The <i>write()</i> and <i>pwrite()</i> functions may fail if:
50901 XSR	[EINVAL]	The STREAM or multiplexer referenced by <i>fildev</i> is linked (directly or indirectly) downstream from a multiplexer.
50902		
50903	[EIO]	A physical I/O error has occurred.
50904	[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
50905	[ENXIO]	A request was made of a nonexistent device, or the request was outside the capabilities of the device.
50906		
50907 XSR	[ENXIO]	A hangup occurred on the STREAM being written to.
50908 XSR		A write to a STREAMS file may fail if an error message has been received at the STREAM head. In this case, <i>errno</i> is set to the value included in the error message.
50909		
50910		The <i>write()</i> function may fail if:
50911	[EACCES]	A write was attempted on a socket and the calling process does not have appropriate privileges.
50912		
50913	[ENETDOWN]	A write was attempted on a socket and the local network interface used to reach the destination is down.
50914		
50915	[ENETUNREACH]	
50916		A write was attempted on a socket and no route to the network is present.
50917 XSI		The <i>pwrite()</i> function shall fail and the file pointer remain unchanged if:
50918 XSI	[EINVAL]	The <i>offset</i> argument is invalid. The value is negative.
50919 XSI	[ESPIPE]	<i>fildev</i> is associated with a pipe or FIFO.

50920 **EXAMPLES**50921 **Writing from a Buffer**

50922 The following example writes data from the buffer pointed to by *buf* to the file associated with
 50923 the file descriptor *fd*.

```
50924 #include <sys/types.h>
50925 #include <string.h>
50926 ...
50927 char buf[20];
50928 size_t nbytes;
50929 ssize_t bytes_written;
50930 int fd;
50931 ...
50932 strcpy(buf, "This is a test\n");
50933 nbytes = strlen(buf);

50934 bytes_written = write(fd, buf, nbytes);
50935 ...
```

50936 **APPLICATION USAGE**

50937 None.

50938 **RATIONALE**

50939 See also the RATIONALE section in *read()*.

50940 An attempt to write to a pipe or FIFO has several major characteristics:

- 50941 • *Atomic/non-atomic*: A write is atomic if the whole amount written in one operation is not
 50942 interleaved with data from any other process. This is useful when there are multiple writers
 50943 sending data to a single reader. Applications need to know how large a write request can be
 50944 expected to be performed atomically. This maximum is called {PIPE_BUF}. This volume of
 50945 IEEE Std 1003.1-2001 does not say whether write requests for more than {PIPE_BUF} bytes
 50946 are atomic, but requires that writes of {PIPE_BUF} or fewer bytes shall be atomic.
- 50947 • *Blocking/immediate*: Blocking is only possible with O_NONBLOCK clear. If there is enough
 50948 space for all the data requested to be written immediately, the implementation should do so.
 50949 Otherwise, the process may block; that is, pause until enough space is available for writing.
 50950 The effective size of a pipe or FIFO (the maximum amount that can be written in one
 50951 operation without blocking) may vary dynamically, depending on the implementation, so it
 50952 is not possible to specify a fixed value for it.

- 50953 • *Complete/partial/deferred*: A write request:

```
50954 int fildes;
50955 size_t nbyte;
50956 ssize_t ret;
50957 char *buf;
```

```
50958 ret = write(fildes, buf, nbyte);
```

50959 may return:

50960 Complete *ret*=*nbyte*

50961 Partial *ret*<*nbyte*

50962 This shall never happen if *nbyte*≤{PIPE_BUF}. If it does happen (with
 50963 *nbyte*>{PIPE_BUF}), this volume of IEEE Std 1003.1-2001 does not guarantee

50964 atomicity, even if $ret \leq \{PIPE_BUF\}$, because atomicity is guaranteed according
 50965 to the amount *requested*, not the amount *written*.

50966 Deferred: $ret = -1$, $errno = [EAGAIN]$

50967 This error indicates that a later request may succeed. It does not indicate that it
 50968 *shall* succeed, even if $nbyte \leq \{PIPE_BUF\}$, because if no process reads from the
 50969 pipe or FIFO, the write never succeeds. An application could usefully count the
 50970 number of times $[EAGAIN]$ is caused by a particular value of
 50971 $nbyte > \{PIPE_BUF\}$ and perhaps do later writes with a smaller value, on the
 50972 assumption that the effective size of the pipe may have decreased.

50973 Partial and deferred writes are only possible with $O_NONBLOCK$ set.

50974 The relations of these properties are shown in the following tables:

50975

50976

50977

50978

50979

50980

Write to a Pipe or FIFO with $O_NONBLOCK$ clear			
Immediately Writable:	None	Some	$nbyte$
$nbyte \leq \{PIPE_BUF\}$	Atomic blocking $nbyte$	Atomic blocking $nbyte$	Atomic immediate $nbyte$
$nbyte > \{PIPE_BUF\}$	Blocking $nbyte$	Blocking $nbyte$	Blocking $nbyte$

50981 If the $O_NONBLOCK$ flag is clear, a write request shall block if the amount writable
 50982 immediately is less than that requested. If the flag is set (by $fctl()$), a write request shall never
 50983 block.

50984

50985

50986

50987

50988

50989

Write to a Pipe or FIFO with $O_NONBLOCK$ set			
Immediately Writable:	None	Some	$nbyte$
$nbyte \leq \{PIPE_BUF\}$	-1, $[EAGAIN]$	-1, $[EAGAIN]$	Atomic $nbyte$
$nbyte > \{PIPE_BUF\}$	-1, $[EAGAIN]$	$< nbyte$ or -1, $[EAGAIN]$	$\leq nbyte$ or -1, $[EAGAIN]$

50990 There is no exception regarding partial writes when $O_NONBLOCK$ is set. With the exception
 50991 of writing to an empty pipe, this volume of IEEE Std 1003.1-2001 does not specify exactly when a
 50992 partial write is performed since that would require specifying internal details of the
 50993 implementation. Every application should be prepared to handle partial writes when
 50994 $O_NONBLOCK$ is set and the requested amount is greater than $\{PIPE_BUF\}$, just as every
 50995 application should be prepared to handle partial writes on other kinds of file descriptors.

50996 The intent of forcing writing at least one byte if any can be written is to assure that each write
 50997 makes progress if there is any room in the pipe. If the pipe is empty, $\{PIPE_BUF\}$ bytes must be
 50998 written; if not, at least some progress must have been made.

50999 Where this volume of IEEE Std 1003.1-2001 requires -1 to be returned and $errno$ set to
 51000 $[EAGAIN]$, most historical implementations return zero (with the O_NDELAY flag set, which is
 51001 the historical predecessor of $O_NONBLOCK$, but is not itself in this volume of
 51002 IEEE Std 1003.1-2001). The error indications in this volume of IEEE Std 1003.1-2001 were chosen
 51003 so that an application can distinguish these cases from end-of-file. While $write()$ cannot receive
 51004 an indication of end-of-file, $read()$ can, and the two functions have similar return values. Also,
 51005 some existing systems (for example, Eighth Edition) permit a write of zero bytes to mean that
 51006 the reader should get an end-of-file indication; for those systems, a return value of zero from
 51007 $write()$ indicates a successful write of an end-of-file indication.

- 51008 Implementations are allowed, but not required, to perform error checking for *write()* requests of
51009 zero bytes.
- 51010 The concept of a {PIPE_MAX} limit (indicating the maximum number of bytes that can be
51011 written to a pipe in a single operation) was considered, but rejected, because this concept would
51012 unnecessarily limit application writing.
- 51013 See also the discussion of O_NONBLOCK in *read()*.
- 51014 Writes can be serialized with respect to other reads and writes. If a *read()* of file data can be
51015 proven (by any means) to occur after a *write()* of the data, it must reflect that *write()*, even if the
51016 calls are made by different processes. A similar requirement applies to multiple write operations
51017 to the same file position. This is needed to guarantee the propagation of data from *write()* calls
51018 to subsequent *read()* calls. This requirement is particularly significant for networked file
51019 systems, where some caching schemes violate these semantics.
- 51020 Note that this is specified in terms of *read()* and *write()*. The XSI extensions *readv()* and *writv()*
51021 also obey these semantics. A new “high-performance” write analog that did not follow these
51022 serialization requirements would also be permitted by this wording. This volume of
51023 IEEE Std 1003.1-2001 is also silent about any effects of application-level caching (such as that
51024 done by *stdio*).
- 51025 This volume of IEEE Std 1003.1-2001 does not specify the value of the file offset after an error is
51026 returned; there are too many cases. For programming errors, such as [EBADF], the concept is
51027 meaningless since no file is involved. For errors that are detected immediately, such as
51028 [EAGAIN], clearly the pointer should not change. After an interrupt or hardware error, however,
51029 an updated value would be very useful and is the behavior of many implementations.
- 51030 This volume of IEEE Std 1003.1-2001 does not specify behavior of concurrent writes to a file from
51031 multiple processes. Applications should use some form of concurrency control.
- 51032 **FUTURE DIRECTIONS**
- 51033 None.
- 51034 **SEE ALSO**
- 51035 *chmod()*, *creat()*, *dup()*, *fcntl()*, *getrlimit()*, *lseek()*, *open()*, *pipe()*, *ulimit()*, *writv()*, the Base
51036 Definitions volume of IEEE Std 1003.1-2001, <limits.h>, <stropts.h>, <sys/uio.h>, <unistd.h>
- 51037 **CHANGE HISTORY**
- 51038 First released in Issue 1. Derived from Issue 1 of the SVID.
- 51039 **Issue 5**
- 51040 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
51041 Threads Extension.
- 51042 Large File Summit extensions are added.
- 51043 The *pwrite()* function is added.
- 51044 **Issue 6**
- 51045 The DESCRIPTION states that the *write()* function does not block the thread. Previously this
51046 said “process” rather than “thread”.
- 51047 The DESCRIPTION and ERRORS sections are updated so that references to STREAMS are
51048 marked as part of the XSI STREAMS Option Group.
- 51049 The following new requirements on POSIX implementations derive from alignment with the
51050 Single UNIX Specification:

- 51051 • The DESCRIPTION now states that if *write()* is interrupted by a signal after it has
51052 successfully written some data, it returns the number of bytes written. In the POSIX.1-1988
51053 standard, it was optional whether *write()* returned the number of bytes written, or whether it
51054 returned `-1` with *errno* set to `[EINTR]`. This is a FIPS requirement.
- 51055 • The following changes are made to support large files:
- 51056 — For regular files, no data transfer occurs past the offset maximum established in the open
51057 file description associated with the *files*.
- 51058 — A second `[EFBIG]` error condition is added.
- 51059 • The `[EIO]` error condition is added.
- 51060 • The `[EPIPE]` error condition is added for when a pipe has only one end open.
- 51061 • The `[ENXIO]` optional error condition is added.
- 51062 Text referring to sockets is added to the DESCRIPTION.
- 51063 The following changes were made to align with the IEEE P1003.1a draft standard:
- 51064 • The effect of reading zero bytes is clarified.
- 51065 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that
51066 *write()* results are unspecified for typed memory objects.
- 51067 The following error conditions are added for operations on sockets: `[EAGAIN]`,
51068 `[EWOULDBLOCK]`, `[ECONNRESET]`, `[ENOTCONN]`, and `[EPIPE]`.
- 51069 The `[EIO]` error is changed to “may fail”.
- 51070 The `[ENOBUFFS]` error is added for sockets.
- 51071 The following error conditions are added for operations on sockets: `[EACCES]`, `[ENETDOWN]`,
51072 and `[ENETUNREACH]`.
- 51073 The *writenv()* function is split out into a separate reference page.

51074 **NAME**

51075 writev — write a vector

51076 **SYNOPSIS**

51077 xSI #include <sys/uio.h>

51078 ssize_t writev(int *fildes*, const struct iovec **iov*, int *iovcnt*);

51079

51080 **DESCRIPTION**

51081 The *writev()* function shall be equivalent to *write()*, except as described below. The *writev()*
 51082 function shall gather output data from the *iovcnt* buffers specified by the members of the *iov*
 51083 array: *iov*[0], *iov*[1], ..., *iov*[*iovcnt*−1]. The *iovcnt* argument is valid if greater than 0 and less than
 51084 or equal to {IOV_MAX}, as defined in <limits.h>.

51085 Each *iovec* entry specifies the base address and length of an area in memory from which data
 51086 should be written. The *writev()* function shall always write a complete area before proceeding to
 51087 the next.

51088 If *fildes* refers to a regular file and all of the *iov_len* members in the array pointed to by *iov* are 0,
 51089 *writev()* shall return 0 and have no other effect. For other file types, the behavior is unspecified.

51090 If the sum of the *iov_len* values is greater than {SSIZE_MAX}, the operation shall fail and no data
 51091 shall be transferred.

51092 **RETURN VALUE**

51093 Upon successful completion, *writev()* shall return the number of bytes actually written.
 51094 Otherwise, it shall return a value of −1, the file-pointer shall remain unchanged, and *errno* shall
 51095 be set to indicate an error.

51096 **ERRORS**51097 Refer to *write()*.51098 In addition, the *writev()* function shall fail if:

51099 [EINVAL] The sum of the *iov_len* values in the *iov* array would overflow an *ssize_t*.

51100 The *writev()* function may fail and set *errno* to:

51101 [EINVAL] The *iovcnt* argument was less than or equal to 0, or greater than {IOV_MAX}.

51102 **EXAMPLES**51103 **Writing Data from an Array**

51104 The following example writes data from the buffers specified by members of the *iov* array to the
 51105 file associated with the file descriptor *fd*.

51106 #include <sys/types.h>

51107 #include <sys/uio.h>

51108 #include <unistd.h>

51109 ...

51110 ssize_t bytes_written;

51111 int fd;

51112 char *buf0 = "short string\n";

51113 char *buf1 = "This is a longer string\n";

51114 char *buf2 = "This is the longest string in this example\n";

51115 int iovcnt;

51116 struct iovec iov[3];


```
51117         iov[0].iov_base = buf0;
51118         iov[0].iov_len = strlen(buf0);
51119         iov[1].iov_base = buf1;
51120         iov[1].iov_len = strlen(buf1);
51121         iov[2].iov_base = buf2;
51122         iov[2].iov_len = strlen(buf2);
51123         ...
51124         iovcnt = sizeof(iov) / sizeof(struct iovec);

51125         bytes_written = writenv(fd, iov, iovcnt);
51126         ...
```

51127 APPLICATION USAGE

51128 None.

51129 RATIONALE

51130 Refer to *write()*.

51131 FUTURE DIRECTIONS

51132 None.

51133 SEE ALSO

51134 *readv()*, *write()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**limits.h**>, <**sys/uio.h**>

51135 CHANGE HISTORY

51136 First released in Issue 4, Version 2.

51137 Issue 6

51138 Split out from the *write()* reference page.

51139 **NAME**

51140 wscanf — convert formatted wide-character input

51141 **SYNOPSIS**

51142 #include <stdio.h>

51143 #include <wchar.h>

51144 int wscanf(const wchar_t *restrict *format*, ...);51145 **DESCRIPTION**51146 Refer to *fwscanf()*.

51147 **NAME**

51148 y0, y1, yn — Bessel functions of the second kind

51149 **SYNOPSIS**

```
51150 xSI      #include <math.h>
51151          double y0(double x);
51152          double y1(double x);
51153          double yn(int n, double x);
51154
```

51155 **DESCRIPTION**

51156 The *y0()*, *y1()*, and *yn()* functions shall compute Bessel functions of *x* of the second kind of
 51157 orders 0, 1, and *n*, respectively.

51158 An application wishing to check for error situations should set *errno* to zero and call
 51159 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
 51160 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
 51161 zero, an error has occurred.

51162 **RETURN VALUE**

51163 Upon successful completion, these functions shall return the relevant Bessel value of *x* of the
 51164 second kind.

51165 If *x* is NaN, NaN shall be returned.

51166 If the *x* argument to these functions is negative, *–HUGE_VAL* or NaN shall be returned, and a
 51167 domain error may occur.

51168 If *x* is 0.0, *–HUGE_VAL* shall be returned and a range error may occur.

51169 If the correct result would cause underflow, 0.0 shall be returned and a range error may occur.

51170 If the correct result would cause overflow, *–HUGE_VAL* or 0.0 shall be returned and a range
 51171 error may occur.

51172 **ERRORS**

51173 These functions may fail if:

- | | | |
|-------|--------------|---|
| 51174 | Domain Error | The value of <i>x</i> is negative. |
| 51175 | | If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
| 51176 | | then <i>errno</i> shall be set to [EDOM]. If the integer expression (math_errhandling |
| 51177 | | & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception |
| 51178 | | shall be raised. |
| 51179 | Range Error | The value of <i>x</i> is 0.0, or the correct result would cause overflow. |
| 51180 | | If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
| 51181 | | then <i>errno</i> shall be set to [ERANGE]. If the integer expression |
| 51182 | | (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow |
| 51183 | | floating-point exception shall be raised. |
| 51184 | Range Error | The value of <i>x</i> is too large in magnitude, or the correct result would cause |
| 51185 | | underflow. |
| 51186 | | If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, |
| 51187 | | then <i>errno</i> shall be set to [ERANGE]. If the integer expression |
| 51188 | | (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow |
| 51189 | | floating-point exception shall be raised. |

51190 EXAMPLES

51191 None.

51192 APPLICATION USAGE

51193 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
51194 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

51195 RATIONALE

51196 None.

51197 FUTURE DIRECTIONS

51198 None.

51199 SEE ALSO

51200 *feclearexcept()*, *fetetestexcept()*, *isnan()*, *j0()*, the Base Definitions volume of IEEE Std 1003.1-2001,
51201 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

51202 CHANGE HISTORY

51203 First released in Issue 1. Derived from Issue 1 of the SVID.

51204 Issue 5

51205 The DESCRIPTION is updated to indicate how an application should check for an error. This
51206 text was previously published in the APPLICATION USAGE section.

51207 Issue 6

51208 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

51209 The RETURN VALUE and ERRORS sections are reworked for alignment of the error handling
51210 with the ISO/IEC 9899:1999 standard.

Index

<code>_CS_PATH</code>	212	<code>_POSIX2 constants</code>	
<code>_CS_XBS5_ILP32_OFF32_CFLAGS</code>	212	in <code>sysconf</code>	1466
<code>_CS_XBS5_ILP32_OFF32_LDFLAGS</code>	212	<code>_POSIX2_CHAR_TERM</code>	1468
<code>_CS_XBS5_ILP32_OFF32_LIBS</code>	212	<code>_POSIX2_C_BIND</code>	1468
<code>_CS_XBS5_ILP32_OFFBIG_CFLAGS</code>	212	<code>_POSIX2_C_DEV</code>	1468
<code>_CS_XBS5_ILP32_OFFBIG_LDFLAGS</code>	212	<code>_POSIX2_C_VERSION</code>	1468
<code>_CS_XBS5_ILP32_OFFBIG_LIBS</code>	212	<code>_POSIX2_FORT_DEV</code>	1468
<code>_CS_XBS5_LP64_OFF64_CFLAGS</code>	212	<code>_POSIX2_FORT_RUN</code>	1468
<code>_CS_XBS5_LP64_OFF64_LDFLAGS</code>	212	<code>_POSIX2_LOCALEDEF</code>	1468
<code>_CS_XBS5_LP64_OFF64_LIBS</code>	212	<code>_POSIX2_PBS</code>	1468
<code>_CS_XBS5_LPBIG_OFFBIG_CFLAGS</code>	212	<code>_POSIX2_PBS_ACCOUNTING</code>	1468
<code>_CS_XBS5_LPBIG_OFFBIG_LDFLAGS</code>	212	<code>_POSIX2_PBS_LOCATE</code>	1468
<code>_CS_XBS5_LPBIG_OFFBIG_LIBS</code>	212	<code>_POSIX2_PBS_MESSAGE</code>	1468
<code>_exit</code>	85, 305, 1585	<code>_POSIX2_PBS_TRACK</code>	1468
<code>_Exit()</code>	305, 85	<code>_POSIX2_SW_DEV</code>	1468
<code>_FILE</code>	127	<code>_POSIX2_UPE</code>	1468
<code>_IOFBF</code>	1279, 1321	<code>_POSIX2_VERSION</code>	1468
<code>_IOLBF</code>	374, 1321	<code>_POSIX</code>	15, 17
<code>_IONBF</code>	1279, 1321	<code>_POSIX_ADVISORY_INFO</code>	1467
<code>_LINE</code>	127	<code>_POSIX_ASYNCHRONOUS_IO</code>	1467
<code>_longjmp()</code>	86	<code>_POSIX_ASYNC_IO</code>	397
<code>_LVL</code>	17	<code>_POSIX_BARRIERS</code>	1467
<code>_MAX</code>	16	<code>_POSIX_CHOWN_RESTRICTED</code>	188, 397, 399
<code>_MIN</code>	16	<code>_POSIX_CLOCK_SELECTION</code>	1467
<code>_PC constants</code>		<code>_POSIX_CPUTIME</code>	1467
used in <code>pathconf</code>	397	<code>_POSIX_C_SOURCE</code>	14
<code>_PC_ALLOC_SIZE_MIN</code>	397	<code>_POSIX_FILE_LOCKING</code>	1467
<code>_PC_ASYNC_IO</code>	397	<code>_POSIX_FSYNC</code>	1467
<code>_PC_CHOWN_RESTRICTED</code>	397	<code>_POSIX_JOB_CONTROL</code>	1467
<code>_PC_FILESIZEBITS</code>	397	<code>_POSIX_MAPPED_FILES</code>	1467
<code>_PC_LINK_MAX</code>	397	<code>_POSIX_MEMLOCK</code>	1467
<code>_PC_MAX_CANON</code>	397	<code>_POSIX_MEMLOCK_RANGE</code>	1467
<code>_PC_MAX_INPUT</code>	397	<code>_POSIX_MEMORY_PROTECTION</code>	1467
<code>_PC_NAME_MAX</code>	397	<code>_POSIX_MESSAGE_PASSING</code>	1467
<code>_PC_NO_TRUNC</code>	397	<code>_POSIX_MONOTONIC_CLOCK</code>	1467
<code>_PC_PATH_MAX</code>	397	<code>_POSIX_MULTI_PROCESS</code>	1467
<code>_PC_PIPE_BUF</code>	397	<code>_POSIX_NO_TRUNC</code>	397
<code>_PC_PRIO_IO</code>	397	<code>_POSIX_OPEN_MAX</code>	553
<code>_PC_REC_INCR_XFER_SIZE</code>	397	<code>_POSIX_PRIORITIZED_IO</code>	42, 1467
<code>_PC_REC_MAX_XFER_SIZE</code>	397	<code>_POSIX_PRIORITY_SCHEDULING</code>	42, 1467
<code>_PC_REC_MIN_XFER_SIZE</code>	397	<code>_POSIX_PRIO_IO</code>	397
<code>_PC_REC_XFER_ALIGN</code>	397	<code>_POSIX_READER_WRITER_LOCKS</code>	1467
<code>_PC_SYMLINK_MAX</code>	397	<code>_POSIX_REALTIME_SIGNALS</code>	1467
<code>_PC_SYNC_IO</code>	397	<code>_POSIX_REGEX</code>	1467
<code>_PC_VDISABLE</code>	397	<code>_POSIX_SAVED_IDS</code>	1467
		<code>_POSIX_SEMAPHORES</code>	1467

_POSIX_SHARED_MEMORY_OBJECTS	1467	_SC_2_PBS_MESSAGE	1468
_POSIX_SHELL	1467	_SC_2_PBS_TRACK	1468
_POSIX_SOURCE	14	_SC_2_SW_DEV	1468
_POSIX_SPAWN	1467	_SC_2_UPE	1468
_POSIX_SPIN_LOCKS	1467	_SC_2_VERSION	860, 1468
_POSIX_SPORADIC_SERVER	1467	_SC_ADVISORY_INFO	1467
_POSIX_SYNCHRONIZED_IO	1467	_SC_AIO_LISTIO_MAX	1466
_POSIX_SYNC_IO	397	_SC_AIO_MAX	1466
_POSIX_THREADS	239, 1467, 1526	_SC_AIO_PRIO_DELTA_MAX	1466
_POSIX_THREAD_ATTR_STACKADDR	1467	_SC_ARG_MAX	1466
_POSIX_THREAD_ATTR_STACKSIZE	1467	_SC_ASYNCHRONOUS_IO	1467
_POSIX_THREAD_CPUTIME	1467	_SC_ATEXIT_MAX	1466
_POSIX_THREAD_PRIORITY_SCHEDULING	1467	_SC_BARRIERS	1467
_POSIX_THREAD_PRIO_INHERIT	1467	_SC_BC_BASE_MAX	1466
_POSIX_THREAD_PRIO_PROTECT	1467	_SC_BC_DIM_MAX	1466
_POSIX_THREAD_PROCESS_SHARED	1088	_SC_BC_SCALE_MAX	1466
.....	1467	_SC_BC_STRING_MAX	1466
_POSIX_THREAD_SAFE_FUNCTIONS	239	_SC_CHILD_MAX	1466
.....	1467, 1526	_SC_CLK_TCK	1466, 1522
_POSIX_THREAD_SPORADIC_SERVER	1467	_SC_CLOCK_SELECTION	1467
_POSIX_TIMEOUTS	1467	_SC_COLL_WEIGHTS_MAX	1466
_POSIX_TIMERS	1467	_SC_CPUTIME	1467
_POSIX_TRACE	1467	_SC_DELAYTIMER_MAX	1466
_POSIX_TRACE_EVENT_FILTER	1467	_SC_EXPR_NEST_MAX	1466
_POSIX_TRACE_EVENT_NAME_MAX	936, 938	_SC_FILE_LOCKING	1467
_POSIX_TRACE_INHERIT	1467	_SC_FSYNC	1467
_POSIX_TRACE_LOG	1467	_SC_GETGR_R_SIZE_MAX	505, 508, 1466
_POSIX_TRACE_SYS_MAX	933	_SC_GETPW_R_SIZE_MAX	545, 548, 1466
_POSIX_TRACE_USER_EVENT_MAX	938	_SC_IOV_MAX	1466
_POSIX_TYPED_MEMORY_OBJECTS	1467	_SC_JOB_CONTROL	1467
_POSIX_V6_ILP32_OFF32	1468	_SC_LINE_MAX	1466
_POSIX_V6_ILP32_OFFBIG	1468	_SC_LOGIN_NAME_MAX	1466
_POSIX_V6_LP64_OFF64	1468	_SC_MEMLOCK	1467
_POSIX_V6_LPBIG_OFFBIG	1468	_SC_MEMLOCK_RANGE	1467
_POSIX_VDISABLE	397	_SC_MEMORY_PROTECTION	1467
_POSIX_VERSION	1468, 1550	_SC_MESSAGE_PASSING	1467
_PROCESS	17	_SC_MONOTONIC_CLOCK	1467
_PTHREAD_THREADS_MAX	1062	_SC_MQ_OPEN_MAX	1466
_REGEX_VERSION	1468	_SC_MQ_PRIO_MAX	1466
_SC constants		_SC_MULTI_PROCESS	1467
in sysconf	1466	_SC_NGROUPS_MAX	1466
_SC_2_CHAR_TERM	1468	_SC_OPEN_MAX	1467
_SC_2_C_BIND	1468	_SC_PAGESIZE	773, 865, 1468
_SC_2_C_DEV	1468	_SC_PAGE_SIZE	773, 1468
_SC_2_C_VERSION	1468	_SC_PRIORITIZED_IO	1467
_SC_2_FORT_DEV	1468	_SC_PRIORITY_SCHEDULING	1467
_SC_2_FORT_RUN	1468	_SC_READER_WRITER_LOCKS	1467
_SC_2_LOCALEDEF	1468	_SC_REALTIME_SIGNALS	1467
_SC_2_PBS_ACCOUNTING	1468	_SC_REGEX	1467
_SC_2_PBS_LOCATE	1468	_SC_REGEX_VERSION	1468
		_SC_RE_DUP_MAX	1468

_SC_RTSIG_MAX	1468	_SC_XOPEN_REALTIME.....	1469
_SC_SAVED_IDS	1467	_SC_XOPEN_REALTIME_THREADS.....	1469
_SC_SEMAPHORES	1467	_SC_XOPEN_SHM	1469
_SC_SEM_NSEMS_MAX.....	1468	_SC_XOPEN_UNIX	1469
_SC_SEM_VALUE_MAX.....	1468	_SC_XOPEN_VERSION	1469
_SC_SHARED_MEMORY_OBJECTS	1467	_SC_XOPEN_XCU_VERSION	1469
_SC_SHELL	1467	_setjmp.....	86
_SC_SIGQUEUE_MAX	1468	_t	17
_SC_SPAWN	1467	_TIME.....	17
_SC_SPIN_LOCKS.....	1467	_tolower()	88
_SC_SPORADIC_SERVER.....	1467	_toupper().....	89
_SC_STREAM_MAX	1468	_XBS5_ILP32_OFF32	1468
_SC_SYMLINK_MAX.....	1468	_XBS5_ILP32_OFFBIG.....	1468
_SC_SYNCHRONIZED_IO	1467	_XBS5_LP64_OFF64.....	1468
_SC_THREADS.....	1467	_XBS5_LPBIG_OFFBIG.....	1468
_SC_THREAD_ATTR_STACKADDR.....	1467	_XOPEN_CRYPT.....	1468
_SC_THREAD_ATTR_STACKSIZE	1467	_XOPEN_ENH_I18N.....	1468
_SC_THREAD_CPUTIME.....	1467	_XOPEN_LEGACY	1469
_SC_THREAD_DESTRUCTOR_ITERATIONS.....	1468	_XOPEN_REALTIME.....	335, 1469
_SC_THREAD_KEYS_MAX	1468	_XOPEN_REALTIME_THREADS	1469
_SC_THREAD_PRIORITY_SCHEDULING.....	1467	_XOPEN_SHM.....	1469
_SC_THREAD_PRIO_INHERIT	1467	_XOPEN_SOURCE	14
_SC_THREAD_PRIO_PROTECT	1467	_XOPEN_UNIX	1469
_SC_THREAD_PROCESS_SHARED.....	1467	_XOPEN_VERSION.....	1469
_SC_THREAD_SAFE_FUNCTIONS	1467	_XOPEN_XCU_VERSION	1469
_SC_THREAD_SPORADIC_SERVER	1467	a64l()	90
_SC_THREAD_STACK_MIN	1468	ABDAY_1	831
_SC_THREAD_THREADS_MAX.....	1468	abort()	92
_SC_TIMEOUTS.....	1467	abs().....	93
_SC_TIMERS	1467	accept()	94
_SC_TIMER_MAX	1468	access()	96
_SC_TRACE.....	1467	acos().....	99
_SC_TRACE_EVENT_FILTER	1467	acosf()	99
_SC_TRACE_INHERIT.....	1467	acosh()	101
_SC_TRACE_LOG	1467	acoshf().....	101
_SC_TTY_NAME_MAX	1468	acoshl()	101
_SC_TYPED_MEMORY_OBJECTS	1467	acosl()	99, 103
_SC_TZNAME_MAX	1468	ACTION	581
_SC_V6_ILP32_OFF32.....	1468	address information.....	424
_SC_V6_ILP32_OFFBIG.....	1468	address string	424
_SC_V6_LP64_OFF64.....	1468	addrinfo structure	424
_SC_V6_LPBIG_OFFBIG	1468	ADV	3
_SC_VERSION.....	1468	ADVANCED REALTIME.....	194, 198, 797-798, 861
_SC_XBS5_ILP32_OFF32.....	1468	863, 865, 867, 869, 872, 880, 883
_SC_XBS5_ILP32_OFFBIG	1468	885-887, 889, 891, 893, 895, 897
_SC_XBS5_LP64_OFF64.....	1468	899, 901-908, 961, 963, 1039
_SC_XBS5_LPBIG_OFFBIG.....	1468	1084, 1253
_SC_XOPEN_CRYPT	1468	ADVANCED REALTIME THREADS.....	1008, 1010
_SC_XOPEN_ENH_I18N	1468	1012, 1014, 1016-1017, 1056, 1137
_SC_XOPEN_LEGACY.....	1469	1139, 1141
		AF.....	18

AIO	3	atoi()	138
AIO_	16	atol()	140
aio	16	atoll()	140
AIO_ALLDONE	104	attributes, clock-resolution	76, 911
aio_cancel()	104	attributes, creation-time	76, 911
AIO_CANCELED	104	attributes, generation-version	75, 911
aio_error()	106	attributes, inheritance	76, 913
aio_fsync()	107	attributes, log-full-policy	74, 76, 913, 916
AIO_LISTIO_MAX	684, 1466	attributes, log-max-size	76, 914, 916
AIO_MAX	684, 1466	attributes, max-data-size	76, 916-917
AIO_NOTCANCELED	104	attributes, stream-full-policy	72-73, 76, 914
AIO_PRIO_DELTA_MAX	42, 1466	attributes, stream-min-size	76, 917
aio_read()	109	attributes, trace-name	76, 911
aio_return()	112	attributes, truncation-status	936
aio_suspend()	113	background	1299
aio_write()	115	background process	1497
AI_ALL	425	BAR	3
AI_CANONNAME	425	basename()	141
AI_INET6	425	baud rate functions	178
AI_NUMERICHOST	425	bcmp()	143
AI_NUMERICSERV	425	bcopy()	144
AI_PASSIVE	425	BC_ constants	
AI_V4MAPPED	425	in sysconf	1466
alarm()	118	BC_BASE_MAX	1466
anycast	67	BC_DIM_MAX	1466
ANYMARK	617	BC_SCALE_MAX	1466
appropriate privileges	97, 398	BC_STRING_MAX	1466
argc	300	BE	4
ARG_MAX	22, 295, 298, 302, 1466	bind()	145
asctime()	120	bi	16
asctime_r()	120	BOOT_TIME	282-283
asin()	122	broadcasting a condition	1026
asinf()	122	BSD	118, 189, 308, 332, 341, 399
asinh()	124	536, 666, 753, 820, 1171, 1211
asinhf()	124	1219, 1299, 1342, 1366, 1489
asinhf()	124	1512, 1550, 1585
asinhl()	124	bsd_signal()	147
asinl()	122, 126	bsearch()	149
assert()	127	btowc()	151
async-signal-safe	976	buffer cache	450
atan()	128	BUFSIZ	1279
atan2()	130	BUS_	18
atan2f()	130	byte-oriented stream	37
atan2l()	130	byte-stream mode	1168
atanf()	128, 132	bzero()	152
atanh()	133	cabs()	153
atanhf()	133	cabsf()	153
atanhl()	133	cabsl()	153
atanl()	128, 135	cacos()	154
atexit()	136	cacosf()	154
ATEXIT_MAX	136, 1466	cacosh()	155
atof()	137		

<code>cacoshf()</code>	155	<code>cfsetospeed()</code>	182
<code>cacoshl()</code>	155	change current working directory	184
<code>cacosl()</code>	154, 156	change file modes	187
<code>calloc()</code>	157	change owner and group of file	189
<code>can</code>	1	<code>CHAR_MAX</code>	694, 696
cancel-safe	1129	<code>chdir()</code>	183
cancelability state	54, 1063, 1129	<code>CHILD_MAX</code>	393, 1466
cancelability type	1063, 1129	<code>chmod()</code>	185
cancellation cleanup handler	1023, 1034	<code>chown()</code>	188
.....	1052, 1067	<code>cimag()</code>	191
cancellation points	55	<code>cimagf()</code>	191
canceling execution of a thread	1018	<code>cimagl()</code>	191
canonical name	425	<code>CLD_</code>	18
<code>carg()</code>	159	<code>clearerr()</code>	192
<code>cargf()</code>	159	clock tick	118, 1470, 1522
<code>cargl()</code>	159	clock ticks/second	1466
<code>casin()</code>	160	<code>clock()</code>	193
<code>casinf()</code>	160	clock-resolution attribute	76, 911
<code>casinh()</code>	161	<code>CLOCKS_PER_SEC</code>	193
<code>casinhf()</code>	161	<code>CLOCK_</code>	17
<code>casinhl()</code>	161	<code>clock_</code>	17
<code>casinl()</code>	160, 162	<code>clock_getcpuclockid()</code>	194
<code>catan()</code>	163	<code>clock_getres()</code>	195
<code>catanf()</code>	163	<code>clock_gettime()</code>	195
<code>catanh()</code>	164	<code>CLOCK_MONOTONIC</code>	49, 199
<code>catanhf()</code>	164	<code>clock_nanosleep()</code>	198
<code>catanhl()</code>	164	<code>CLOCK_PROCESS_CPUTIME_ID</code>	50
<code>catanl()</code>	163, 165	<code>CLOCK_REALTIME</code>	49, 195, 199
<code>catclose()</code>	166	820, 1084, 1253, 1514
<code>catgets()</code>	167	<code>clock_settime()</code>	195, 201
<code>catopen()</code>	169	<code>CLOCK_THREAD_CPUTIME_ID</code>	50
<code>CBAUD</code>	18	<code>clog()</code>	202
<code>cbrt()</code>	171	<code>clogf()</code>	202
<code>cbrtf()</code>	171	<code>clogl()</code>	202
<code>cbrtl()</code>	171	close a file	205
<code>ccos()</code>	172	<code>close()</code>	203
<code>ccosf()</code>	172	<code>closedir()</code>	206
<code>ccosh()</code>	173	<code>closelog()</code>	208
<code>ccoshf()</code>	173	<code>cmsg_</code>	16
<code>ccoshl()</code>	173	<code>CMSG_</code>	18
<code>ccosl()</code>	172, 174	<code>COLL_WEIGHTS_MAX</code>	1466
<code>CD</code>	4	command interpreter	
<code>ceil()</code>	175	portable	1585
<code>ceilf()</code>	175	compare thread IDs	1051
<code>ceill()</code>	175	compilation environment	13
<code>cexp()</code>	177	condition variable initialization attributes	1037
<code>cexpf()</code>	177	conforming application	1383
<code>cexpl()</code>	177	conforming application, strictly	118, 300
<code>cfgetispeed()</code>	178	<code>confstr()</code>	212
<code>cfgetospeed()</code>	180	<code>conj()</code>	215
<code>cfsetispeed()</code>	181	<code>confj()</code>	215

conj1()	215	ctanf()	236
connect()	216	ctanh()	237
control data	38	ctanhf()	237
control-normal	1168	ctanhl()	237
conversion descriptor	295, 300, 587-590	ctanl()	236, 238
conversion specification	402, 433, 466	ctermid()	239
modified	475, 1418, 1422, 1434	ctime()	241
conversion specifier	1424	ctime_r()	241
modified	1435	CX	4
copysign()	219	c_	17
copysignf()	219	data key creation	1067
copysignl()	219	data messages	38
core	1586	data type	80
core file	307	DATMSK	493
cos()	220	daylight	243 , 1542
cosf()	220	DBL_MANT_DIG	175, 375
cosh()	222	DBL_MAX_EXP	175, 375
coshf()	222	DBM	244-245
coshl()	222	dbm_	16
cosl()	220, 224	DBM_	18
covert channel	666	dbm_clearerr()	244
cpow()	225	dbm_close()	244
cpowf()	225	dbm_delete()	244
cpowl()	225	dbm_error()	244
cproj()	226	dbm_fetch()	244
cprojf()	226	dbm_firstkey()	244
cprojl()	226	DBM_INSERT	244
CPT	4	dbm_nextkey()	244
creal()	227	dbm_open()	244
crealf()	227	DBM_REPLACE	244
creall()	227	dbm_store()	244
creat()	228	DEAD_PROCESS	282-283
create a per-process timer	1515	DEFCHO	18
create an interprocess channel	853	deferred cancelability	1063
create session and set process group ID	1311	delay process execution	1382
creation-time attribute	76, 911	DELAUTIMER_MAX	1466, 1519
CRYPT	230, 268, 1292	dependency ordering	257
crypt()	230	descriptive name	424
CS	4	destroying a mutex	1073
csin()	232	destroying condition variables	1030
csinf()	232	destructor functions	1067
csinh()	233	detaching a thread	1049
csinhf()	233	diffime()	248
csinhl()	233	DIR	80, 206, 842, 1174, 1176, 1214, 1240, 1505
csinl()	232	directive	402, 433, 466, 475, 1434
csinl()	234	directory operations	843
csqrt()	235	dirent structure	843
csqrtf()	235	dirname()	249
csqrtl()	235	div()	251
ctan()	236	dlclose()	252
		dllerror()	254

dlopen()	256	EMPTY	283
dlsym()	259	EMSGSIZE	24
dot	843, 1211	EMULTIHOP	25
dot-dot	843, 1211	ENAMETOOLONG	25
drand48()	261	encrypt()	268
dup()	263	endgrent()	270
dup2()	263	endhostent()	272
dynamic package initialization	1105	endnetent()	274
d_	16	endprotoent()	276
E2BIG	22	endpwent()	278
EACCES	22	endservent()	280
EADDRINUSE	22	endutxent()	282
EADDRNOTAVAIL	22	ENETDOWN	25
EAFNOSUPPORT	22	ENETRESET	25
EAGAIN	22, 28	ENETUNREACH	25
EALREADY	22	ENFILE	25
EBADF	22	ENOBUFS	25
EBADMSG	22	ENODATA	25
EBUSY	23	ENODEV	25
ECANCELED	23	ENOENT	25
ECHILD	23	ENOEXEC	25
ECHOCTL	18	ENOLCK	25
ECHOKE	18	ENOLINK	25
ECHOPRT	18	ENOMEM	25
ECONNABORTED	23	ENOMSG	25
ECONNREFUSED	23	ENOPROTOPT	26
ECONNRESET	23	ENOSPC	26
ecvt()	266	ENOSR	26
EDEADLK	23	ENOSTR	26
EDESTADDRREQ	23	ENOSYS	26
EDOM	23	ENOTCONN	26
EDQUOT	23	ENOTDIR	26
EEXIST	23	ENOTEMPTY	26
EFAULT	23	ENOTSOCK	26
EFBIG	23	ENOTSUP	26
effective group ID	189, 302, 511	ENOTTY	26
effective user ID	97, 302, 666	ENTRY	581
EHOSTUNREACH	24	environ	285, 301
EIDRM	24	envp	301
Eighth Edition UNIX	1654	ENXIO	26
EILSEQ	24, 38	EOPNOTSUPP	26
EINPROGRESS	24, 43	EOVERFLOW	26
EINTR	24, 57, 930	EPERM	27
EINVAL	24, 917, 930	PIPE	27
EIO	24	EPROTO	27
EISCONN	24	EPROTONOSUPPORT	27
EISDIR	24	EPROTOTYPE	27
ELOOP	24	erand48()	261, 286
ELSIZE	723	ERANGE	27
EMFILE	24	erf()	287
EMLINK	24	erfc()	289

erfcf()	289	extensions to setlocale	1294
erfcl()	289	fabs()	316
erff()	287, 291	fabsf()	316
erfl()	287, 291	fabsl()	316
EROFS	27	fattach()	317
errno	292	fchdir()	320
error descriptions	481	fchmod()	321
error numbers	21	fchown()	323
additional	28	fclose()	325
ESPIPE	27	fcntl()	327
ESRCH	27	fcvt()	266, 334
EST5EDT	1542	FD	4
establishing cancelation handlers	1023	fdatasync()	335
ESTALE	27	fdetach()	336
ETIME	27	fdim()	338
ETIMEDOUT	27	fdimf()	338
ETXTBSY	27	fdiml()	338
EWOULDBLOCK	28	fdopen()	340
examine and change blocked signals	1136	fds	16
examine and change signal action	1342	fd	16
EXDEV	28	FD	16, 18
exec	294	FD_CLOEXEC	35, 169, 295, 327, 590
of shell scripts	300		835, 843, 853, 873, 880, 1323
exec family	97, 205, 331, 372, 395, 976, 1300, 1585	FD_CLR	971
execl	294	FD_CLR()	84
execle	294	FD_ISSET	84, 971
execlp	294	FD_SET	84, 971
execute a file	300	FD_ZERO	84, 971
execution time monitoring	50	feature test macro	13, 486
execv	294	_POSIX_C_SOURCE	14
execve	294	_XOPEN_SOURCE	14
execvp	294	feclearexcept()	342
exit()	305	fegetenv()	343
EXIT_FAILURE	305	fegetexceptflag()	344
EXIT_SUCCESS	305, 308	fegetround()	345
exp()	310	feholdexcept()	347
exp2()	312	feof()	348
exp2f()	312	feraiseexcept()	349
exp2l()	312	ferror()	350
expf()	310	fesetenv	343
expl()	310	fesetenv()	351
expm1()	314	fesetexceptflag()	344, 352
expm1f()	314	fesetround()	345, 353
expm1l()	314	fetestexcept()	354
EXPR_NEST_MAX	1466	feupdateenv()	356
EXTA	18	fflush()	358
EXTB	18	ffs()	361
extension		fgetc()	362
CX	4	fgetpos()	364
OH	6	fgets()	366
XSI	10	fgetwc()	368

<code>fgetws()</code>	370	<code>FNМ_PATHNAME</code>	386
<code>FIFO</code>	755-756, 839, 1653	<code>FNМ_PERIOD</code>	386
<code>FILE</code>	80, 192, 325, 340, 348, 350	<code>fopen()</code>	388
.....	358, 362, 364, 366, 368, 370	<code>FOPEN_MAX</code>	340, 389, 859, 1524
.....	372-373, 388, 402, 414, 416, 418	<code>foreground</code>	1299
.....	420-421, 428, 433, 440, 443, 452	<code>fork handler</code>	977
.....	464-466, 473, 475, 484-485, 570	<code>fork()</code>	392
.....	849, 859, 1144-1145, 1157, 1213	<code>forkall</code>	395
.....	1279, 1321, 1403, 1524, 1552-1553	<code>format of entries</code>	11
.....	1570, 1572, 1574, 1576, 1578, 1580	<code>fpathconf()</code>	397
<code>file</code>		<code>fpclassify()</code>	401
<code>locking</code>	330	<code>FPE_</code>	18
<code>file accessibility</code>	97	<code>fprintf()</code>	402
<code>file control</code>	330	<code>fputc()</code>	414
<code>FILE object</code>	34	<code>fputs()</code>	416
<code>file permission bits</code>	97	<code>fputwc()</code>	418
<code>file permissions</code>	97, 399, 1400	<code>fputws()</code>	420
<code>file position indicator</code>	34	<code>FQDN</code>	526
<code>fileno()</code>	372	<code>FR</code>	4
<code>FILESIZEBITS</code>	397	<code>fread()</code>	421
<code>FIND</code>	581	<code>free()</code>	423
<code>find string token</code>	1447	<code>freeaddrinfo()</code>	424
<code>flockfile()</code>	373	<code>freopen()</code>	428
<code>floor()</code>	375	<code>frexp()</code>	431
<code>floorf()</code>	375	<code>frexpf()</code>	431
<code>floorl()</code>	375	<code>frexpl()</code>	431
<code>FLT_RADIX</code>	713	<code>FSC</code>	5
<code>FLT_ROUNDS</code>	377	<code>fscanf()</code>	433
<code>FLUSH</code>	18	<code>fseek()</code>	440
<code>FLUSHO</code>	18	<code>fseeko()</code>	440
<code>FLUSHR</code>	611	<code>fsetpos()</code>	443
<code>FLUSHRW</code>	611	<code>fstat()</code>	445
<code>FLUSHW</code>	611	<code>fstatvfs()</code>	447
<code>fma()</code>	377	<code>fsync()</code>	450
<code>fmaf()</code>	377	<code>ftell()</code>	452
<code>fmal()</code>	377	<code>ftello()</code>	452
<code>fmax()</code>	379	<code>ftime()</code>	454
<code>fmaxf()</code>	379	<code>ftok()</code>	456
<code>fmaxl()</code>	379	<code>ftruncate()</code>	458
<code>fmin()</code>	380	<code>ftrylockfile()</code>	373, 460
<code>fminf()</code>	380	<code>FTW</code>	18, 826-827
<code>fminl()</code>	380	<code>ftw()</code>	461
<code>FMNAMESZ</code>	610	<code>FTW_CHDIR</code>	826
<code>fmod()</code>	381	<code>FTW_D</code>	461, 826
<code>fmodf()</code>	381	<code>FTW_DEPTH</code>	826
<code>fmodl()</code>	381	<code>FTW_DNR</code>	461, 826-827
<code>fmsg()</code>	383	<code>FTW_DP</code>	826
<code>fnmatch()</code>	386	<code>FTW_F</code>	461, 826
<code>FNМ_</code>	18	<code>FTW_MOUNT</code>	826
<code>FNМ_NOESCAPE</code>	386	<code>FTW_NS</code>	461, 826-827
<code>FNМ_NOMATCH</code>	386	<code>FTW_PHYS</code>	826

FTW_SL.....	461, 826	getegid()	498
FTW_SLN.....	826	getenv()	301, 499
fully-qualified domain name	526	geteuid()	502
functions	13	getgid()	503
implementation.....	13	getgrent()	270, 504
use.....	13	getgrgid()	505
funlockfile()	373, 464	getgrgid_r()	505
fwide()	465	getgrnam()	508
fwprintf()	466	getgrnam_r()	508
fwrite()	473	getgroups()	510
fwscanf()	475	gethostbyaddr()	512
f_	16	gethostbyname()	512
F	18	gethostent()	272
F_DUPFD	263, 327, 329-330	gethostent()	514
F_GETFD	327, 329-330	gethostid()	515
F_GETFL	327, 329-330	gethostname()	516
F_GETLK	328-330	getitimer()	517
F_GETOWN	327, 329	getlogin()	519
F_LOCK	702	getlogin_r	519
F_RDLCK	330	getmsg()	522
F_SETFD	327, 329-331	getnameinfo()	526
F_SETFL	327, 329-330	GETNCNT	1260-1261
F_SETLK	328-330	getnetbyaddr()	274, 528
F_SETLKW	55, 328-330	getnetbyname()	274, 528
F_SETOWN	327, 330	getnetent()	274, 528
F_TEST	702	getopt()	529
F_TLOCK	702	getpeername()	534
F_ULOCK	702	getpgid()	535
F_UNLCK	328-329	getpgrp()	536
F_WRLCK	330	GETPID	1260-1261
gai_strerror()	481	getpid()	537
gcvt()	266, 482	getpmsg()	522, 538
generation-version attribute	75, 911	getppid()	539
get configurable pathname variables	399	getpriority()	540
get configurable system variables.....	1469	getprotent()	543
get file status.....	1400	getprotobyname()	276, 543
get process times	1522	getprotobynumber	276, 543
get supplementary group IDs.....	510	getprotoent()	276
get system time.....	1512	getpwent()	278, 544
get thread ID.....	1127	getpwnam()	545
get user name	520	getpwnam_r()	545
getaddrinfo()	424, 483	getpwuid()	548
GETALL.....	1260	getpwuid_r()	548
getc()	484	getrlimit()	551
getchar()	487	getrusage()	554
getchar_unlocked()	485, 488	gets()	556
getcontext()	489	getservbyname()	280, 557
getcwd()	491	getservbyport()	280, 557
getc_unlocked()	485	getservent()	280, 557
getdate()	493	getsid()	558
getdate_err	493	getsockname()	559

getsockopt()	560	h_	16
getsubopt()	563	h_errno	580
gettimeofday()	567	iconv()	587
getuid()	568	iconv_close()	589
getutxent()	282, 569	iconv_open()	590
getutxid()	282, 569	ic_	16
getutxline()	282, 569	IEEE Std 754-1985	3
GETVAL	1260-1261	IEEE Std 854-1987	3
getwc()	570	ifc_	16
getwchar()	571	ifra_	16
getwd()	492, 572	ifru_	16
GETZCNT	1260-1261	if_	16
glob()	573	IF_	18
globfree()	573	if_freenameindex()	592
GLOB_	18	if_indexoname()	593
GLOB_ constants		if_nameindex()	594
error returns of glob	575	if_nametoindex()	595
used in glob	573	ILL_	18
GLOB_ABORTED	575	ilogb()	596
GLOB_APPEND	573-574	ilogbf()	596
GLOB_DOOFFS	573-574	ilogbl()	596
GLOB_ERR	573, 575	imaxabs()	598
GLOB_MARK	574	imaxdiv()	599
GLOB_NOCHECK	574-575	implementation-defined	1
GLOB_NOESCAPE	574	IMPLINK_	18
GLOB_NOMATCH	575	in6_	16
GLOB_NOSORT	574	IN6_	18
GLOB_NOSPACE	575	INADDR_	18
gl_	16	index()	600
GMT0	1542	inet_	16
gmtime()	577	inet_addr()	601
gmtime_r()	577	inet_ntoa()	601
grantpt()	579	inet_ntop()	603
granularity of clock	454	inet_pton()	603
HALT	384	Inf	122
hcreate()	581	INF	405, 469
hdestroy()	581	INFINITY	405, 469
high resolution sleep	820	INFO	384
host name	424	infu_	16
htonl()	584	inheritance attribute	76, 913
htons()	584	init	308, 666
HUGE_VAL	133, 175, 222, 310, 312	initialize a named semaphore	1249
	314, 338, 375, 585, 674, 678, 705	initialize an unnamed semaphore	1246
	709, 711, 822, 824, 966, 1216	initializing a mutex	1073
	1221, 1225, 1379, 1442, 1478	initializing condition variables	1030
	1509, 1612	initstate()	605
HUGE_VALF	375, 1216, 1442, 1612	INIT_PROCESS	282-283
HUGE_VALL	375, 1216, 1442, 1612	input and output rationale	1170
hypot()	585	insque()	607
hypotf()	585	INT	18
hypotl()	585	international environment	1294

Internet Protocols	66	isastream()	625
INT_MAX	596	isatty()	626
INT_MIN	93	isblank()	627
in_	16	iscntrl()	628
IN_	18	isdigit()	629
ioctl()	610	isfinite()	630
iov_	17	isgraph()	631
IOV_	18	isgreater()	632
IOV_MAX	1181, 1466, 1657	isgreaterequal()	633
IP6	5	isinf()	634
IPC.....	39 , 803, 805, 808, 810, 1265, 1269, 1333, 1335	isless()	635
ipc_	16	islessequal()	636
IPC_	18	islessgreater()	637
IPC_ constants		islower()	638
used in semctl	1260	isnan()	640
used in shmctl	1331	isnormal()	641
IPC_CREAT	804, 1263, 1334	ISO C standard	3, 118, 300, 330
IPC_EXCL	804, 1263	486, 725, 1164, 1211, 1294
IPC_NOWAIT	806-807, 809-810, 1266	1342, 1366, 1512
IPC_PRIVATE	804, 1263, 1334	isprint()	642
IPC_RMID	802, 1261, 1331	ispunct()	643
IPC_SET	802, 1261, 1331	isspace()	644
IPC_STAT	802, 1260, 1331	isunordered()	645
IPPORT_	18	isupper()	646
IPPROTO_	18	iswalnum()	647
IPv4	67	iswalpha()	648
IPv4-compatible address	68	iswblank()	649
IPv4-mapped address	68	iswcntrl()	650
IPv6	67	iswctype()	651
compatibility with IPv4	68	iswdigit()	653
interface identification	68	iswgraph()	654
options	69	iswlower()	655
IPv6 address		iswprint()	656
anycast	67	iswpunct()	657
loopback	68	iswspace()	658
multicast	67	iswupper()	659
unicast	67	iswxdigit()	660
unspecified	68	isxdigit()	661
IPV6_	18	ITIMER_PROF	517
IPV6_JOIN_GROUP	69	ITIMER_REAL	517
IPV6_LEAVE_GROUP	69	ITIMER_VIRTUAL	517
IPV6_MULTICAST_HOPS	69	it_	16-17
IPV6_MULTICAST_IF	69	I_	18
IPV6_MULTICAST_LOOP	69	I_ATMARK	616-617
IPV6_UNICAST_HOPS	69	I_CANPUT	617
IPV6_V6ONLY	69	I_CKBAND	617
ip_	16	I_FDINSERT	613
IP_	18	I_FIND	612
isalnum()	622	I_FLUSH	610
isalpha()	623	I_FLUSHBAND	611
isascii()	624	I_GETBAND	617

I_GETCLTIME	617	ldexpl()	674
I_GETSIG	612	ldiv()	676
I_GRDOPT	613, 1168	legacy	1
I_GWROPT	615	LEGACY	1565
I_LINK	618	lfind()	677 , 723
I_LIST	616	lgamma()	678
I_LOOK	610	lgammaf()	678
I_NREAD	613	lgammal()	678
I_PEEK	612	LIFO	57
I_PLINK	619	LINE_MAX	1466
I_POP	610	link to a file	682
I_PUNLINK	619	link()	680
I_PUSH	610	LINK_MAX	24, 397, 680, 1210
I_RECVFD	22, 616	LIO_	16
I_SENDFD	615-616	lio_	16
I_SETCLTIME	203, 617	lio_listio()	683
I_SETSIG	611-612	LIO_NOP	683
I_SRDOPT	612-613, 1168	LIO_NOWAIT	683
I_STR	614	LIO_READ	683
I_SWROPT	615, 1651	LIO_WAIT	683
I_UNLINK	618	LIO_WRITE	683
j0()	662	list directed I/O	685
j1()	662	listen()	686
jn()	662	llabs()	670, 688
job control	308, 536, 666, 1299, 1311, 1470, 1585	lldiv()	676, 689
jrnd48()	261, 664	LLONG_MAX	1450, 1618
JST-9	1542	LLONG_MIN	1450, 1618
kill()	665	llrint()	690
killpg()	668	llrintf()	690
l64a()	90, 669	llrintl()	690
labs()	670	llround()	692
LANG	169	llroundf()	692
last close	1327	llroundl()	692
LASTMARK	617	load ordering	257
lchown()	671	LOBLK	18
lcong48()	261, 673	localeconv()	694
LC_ALL	295, 696, 831, 1293, 1295	localtime()	699
LC_COLLATE	573-574, 1293-1294	localtime_r()	699
	1410, 1457, 1596, 1630	lockf()	702
LC_CTYPE	151, 651, 733, 735	locking	330
	737, 739-740, 742, 744, 1293-1294	advisory	331
	1529-1533, 1591, 1607, 1622	mandatory	331
	1632-1633, 1635-1636	locking and unlocking a mutex	1081
LC_MESSAGES	169, 1293-1294, 1416	log()	705
LC_MONETARY	696, 1293-1294, 1419	log-full-policy attribute	74, 76, 913, 916, 934
LC_NUMERIC	266, 403, 433, 466	log-max-size attribute	76, 914, 916
	475, 696, 1293-1294, 1419	log10()	707
	1442, 1612	log10f()	707
LC_TIME	494, 831, 1293-1294	log10l()	707
ldexp()	674	log1p()	709
ldexpf()	674	log1pf()	709

log1pl()	709	MAP_FIXED	772, 775
log2()	711	MAP_PRIVATE	392, 772, 776, 780, 812
log2f()	711	MAP_SHARED	395, 772-773
log2l()	711	max-data-size attribute	76, 916-917
logb()	713	MAX_CANON	397
logbf()	713	MAX_INPUT	397
logbl()	713	may	2
logf()	705, 715	mblen()	733
login shell	300	mbrlen()	735
LOGIN_NAME_MAX	519, 1466	mbrtowc()	737
LOGIN_PROCESS	282-283	mbsinit()	739
logl	705, 715	mbsrtowcs()	740
LOG	18	mbstowcs()	742
LOG_constants in syslog	208	mbtowc()	744
LOG_ALERT	208	MB_CUR_MAX	733, 735, 737, 744, 1591, 1633
LOG_CONS	209	MC1	5
LOG_CRIT	208	MC2	5
LOG_DEBUG	208	MCL	16
LOG_EMERG	208	MCL_CURRENT	769
LOG_ERR	208	MCL_FUTURE	769
LOG_INFO	208	memccpy()	746
LOG_LOCAL	208	memchr()	747
LOG_NDELAY	209	memcmp()	748
LOG_NOTICE	208	memcpy()	749
LOG_NOWAIT	209	MEMLOCK_FUTURE	776
LOG_ODELAY	209	memmove()	750
LOG_PID	209	memory management	43
LOG_USER	208-209	memory protection option	775
LOG_WARNING	208	memset()	751
longjmp()	716	message catalog descriptor	295, 300, 305
LONG_MAX	1450, 1618	message parts	39
LONG_MIN	1450, 1618	message priority	39
lrand48()	261, 718	high-priority	39
lrint()	719	normal	39
lrintf()	719	priority	39
lrintl()	719	message-discard mode	1168
lround()	721	message-nondiscard mode	1168
lroundf()	721	MET-1MEST	1542
lroundl()	721	MF	5
lsearch()	723	MINSIGSTKSZ	1345
lseek()	725	mkdir()	752
lstat()	727	mkfifo()	755
l_	16	mknod()	758
L_ctermid	239	mkstemp()	761
l_sysid	331	mktemp()	763
makecontext()	729	mktime()	765
malloc()	731	ML	5
manipulate signal sets	1348	mlock()	767
mappings	776	mlockall()	769
MAP	16, 18	MLR	6
MAP_FAILED	776	mmap()	771

MM_	18	MSG_	18
MM_APPL	383	MSG_ANY	522
MM_CONSOLE	383	MSG_BAND	522, 1150
MM_ERROR	384-385	MSG_EOR	1387, 1389
mm_FIRM	383	MSG_HIPRI	522, 1150
MM_HALT	384	MSG_NOERROR	806-807
MM_HARD	383	msg_perm	40
MM_INFO	384	msqid	40
MM_NOCON	384	MST7MDT	1542
MM_NOMSG	384	msync()	812
MM_NOSEV	384	MS_	16, 18
MM_NOTOK	384	MS_ASYNC	773, 812
MM_NRECOV	383	MS_INVALIDATE	812-813
MM_NULLMC	383	MS_SYNC	773, 812
MM_OK	384	multicast	67
MM_OPSYS	383	munlock()	767, 815
MM_PRINT	383, 385	munlockall()	769, 816
MM_RECOVER	383	munmap()	817
MM_SOFT	383	mutex attributes	1088
MM_UTIL	383	mutex initialization attributes	1087
MM_WARNING	384	mutex performance	1088
modf()	778	MUXID_ALL	618-619
modff()	778	MUXID_R	18
modfl()	778	MX	6
MON	6	M_	18
MORECTL	523	name information	526
MOREDATA	523	name space	14
MPR	6	NAME_MAX	25, 96, 169, 183, 185
mprotect()	780		188, 298, 317, 336, 389, 397
MQ_	16		428, 447, 456, 671, 680, 727
mq_	16		752, 755, 759, 788, 799, 827
mq_close()	782		837, 842, 964, 1174, 1178, 1185
mq_getattr()	783		1210, 1218, 1249, 1257, 1324
mq_notify()	785		1327, 1398, 1463, 1535, 1555
mq_open()	787		1563, 1565
MQ_OPEN_MAX	1466	NaN	122, 405, 469
MQ_PRIO_MAX	793-794, 1466	NAN	405, 469
mq_receive()	790	nan()	819
mq_send()	793	nanf()	819
mq_setattr()	795	nanl()	819
mq_timedreceive()	790, 797	nanosleep()	820
mq_timedsend()	793, 798	NDEBUG	21, 127
mq_unlink()	799	nearbyint()	822
mrnd48()	261, 801	nearbyintf()	822
MSG	6, 18	nearbyintl()	822
msgctl()	802	network interfaces	59
msgget()	804	NEW_TIME	282-283
msgrcv()	806	nextafter()	824
msgsnd()	809	nextafterf()	824
MSGVERB	384-385	nextafterl()	824
msg_	16	nexttoward()	824

nexttowardf()	824	MC2	5
nexttowardl()	824	MF	5
nftw()	826	ML	5
NGROUPS_MAX	511, 1466	MLR	6
nice()	829	MON	6
NLSPATH	169	MPR	6
NL_	18	MSG	6
NL_ARGMAX	402, 433, 466, 475	MX	6
NL_CAT_LOCALE	169	PIO	7
nl_langinfo()	831	PS	7
nohup utility	301	RS	7
non-local jumps	1366	RTS	7
non-volatile storage	450	SD	7
rand48()	261, 833	SEM	7
ntohl()	584, 834	SHM	7
ntohs()	584, 834	SIO	8
NULL	213, 239, 246, 254, 259, 776, 1176	SPI	8
NUM_EMPL	582	SPN	8
NZERO	540, 829	SS	8
n_	16	TCT	8
OB	6	TEF	8
obsolescent	178	THR	8
OF	6	TMO	8
OH	6	TMR	9
OLD_TIME	282-283	TPI	9
open a file	839	TPP	9
open a named semaphore	1249	TPS	9
open a shared memory object	1324	TRC	9
open()	835	TRI	9
opendir()	842	TRL	9
openlog()	208, 845	TSA	10
OPEN_MAX	169, 263, 278, 330, 389	TSF	10
	428, 461, 505, 508, 519, 548, 590	TSH	10
	787, 827, 837, 842, 853, 880, 883	TSP	10
	1467, 1524	TSS	10
optarg	529, 846	TYM	10
opterr	529, 846	UP	10
optind	529, 846	XSR	11
option		optopt	529, 532, 846
ADV	3	optstring	532
AIO	3	orphaned process group	308
BAR	3	O_	18
BE	4	O_ constants	
CD	4	used in open()	835
CPT	4	used in posix_openpt()	870
CS	4	O_ACCMODE	327
FD	4	O_APPEND	42, 115, 244, 341, 835, 1649
FR	4	O_CREAT	228, 787-788, 799, 835-837
FSC	5		1243, 1248, 1323-1325
IP6	5	O_DSYNC	107, 835-836, 1168, 1650
MC1	5	O_EXCL	788, 835-836, 1248, 1323-1324

O_NDELAY	1654	POLLWRBAND	855
O_NOCTTY	836, 840, 870	POLLWRNORM	855
O_NONBLOCK	24, 203, 325, 358, 362	POLL_	18
.....	368, 414, 418, 441, 443, 523	popen()	859
.....	614, 616, 788, 790, 793, 795	portability	3
.....	836-838, 853, 856, 1151, 1167	POSIX	266
.....	1650, 1653	POSIX.1 symbols	13
O_RDONLY	249, 787, 835-837, 840, 1323, 1325	posix_	15, 17
O_RDWR	320, 702, 787, 835-839, 870, 1323, 1325	POSIX_	15, 17
O_RSYNC	836, 1168	POSIX_ALLOC_SIZE_MIN	397
O_SYNC	107, 836, 1168, 1650	posix_fadvise()	861
O_TRUNC	228, 836, 838, 840, 1324-1325	POSIX_FADV_DONTNEED	861
O_WRONLY	228, 320, 702, 787, 835-838, 840	POSIX_FADV_NOREUSE	861
PAGESIZE	43, 767, 813, 817, 983, 1468	POSIX_FADV_NORMAL	861
PAGE_SIZE	1468	POSIX_FADV_RANDOM	861
PATH	213	POSIX_FADV_SEQUENTIAL	861
PATH environment variable	303	POSIX_FADV_WILLNEED	861
pathconf()	397, 847	posix_fallocate()	863
PATH_MAX	25, 96, 169, 183, 185	posix_madvise()	865
.....	188, 249, 298, 317, 336, 389	POSIX_MADV_DONTNEED	865
.....	397, 428, 447, 456, 492, 572	POSIX_MADV_NORMAL	865
.....	671, 680, 727, 752, 755, 759	POSIX_MADV_RANDOM	865
.....	788, 799, 827, 837, 842, 964	POSIX_MADV_SEQUENTIAL	865
.....	1178, 1185, 1210, 1218, 1249	POSIX_MADV_WILLNEED	865
.....	1257, 1324, 1327, 1398, 1463	posix_memalign()	869
.....	1470, 1535, 1555, 1563, 1565	posix_mem_offset()	867
pause()	848	posix_openpt()	870
pclose()	849	POSIX_REC_INCR_XFER_SIZE	397
pd_	16	POSIX_REC_MAX_XFER_SIZE	397
PENDIN	18	POSIX_REC_MIN_XFER_SIZE	397
perror()	851	POSIX_REC_XFER_ALIGN	397
persistent connection (I_PLINK)	619	posix_spawn()	872
PF_	18	posix_spawnattr_destroy()	887
physical write	450	posix_spawnattr_getflags()	889
ph_	16	posix_spawnattr_getpgroup()	891
PIO	7	posix_spawnattr_getschedparam()	893
pipe	394, 840, 1653	posix_spawnattr_getschedpolicy()	895
pipe()	853	posix_spawnattr_getsigdefault()	897
PIPE_BUF	397, 1650, 1653	posix_spawnattr_getsigmask()	899
PIPE_MAX	1655	posix_spawnattr_init()	887, 901
plain characters	1418	posix_spawnattr_setflags()	889, 902
POLL	18	posix_spawnattr_setpgroup()	891, 903
poll()	855	posix_spawnattr_setschedparam()	893, 904
POLLERR	855	posix_spawnattr_setschedpolicy()	895, 905
POLLHUP	855	posix_spawnattr_setsigdefault()	897, 906
POLLIN	855	posix_spawnattr_setsigmask()	899, 907
POLLNVAL	856	posix_spawn()	872, 908
POLLOUT	855	posix_spawn_file_actions_addclose()	880
POLLPRI	855	posix_spawn_file_actions_adddup2()	883
POLLRDBAND	855	posix_spawn_file_actions_addopen()	880, 885
POLLRDNORM	855	posix_spawn_file_actions_destroy()	886

posix_spawn_file_actions_init()	886	POSIX_TRACE_FLUSH	914
POSIX_SPAWN_RESETHIDS	873, 889	posix_trace_flush()	932, 944
POSIX_SPAWN_SETPGROUP	873, 889, 891	POSIX_TRACE_FLUSHING	73
POSIX_SPAWN_SETSCHEDPARAM	889, 893	POSIX_TRACE_FULL	72-74
POSIX_SPAWN_SETSCHEDULER	873, 889	posix_trace_getnext_event()	950
.....	893, 895	posix_trace_get_attr()	945
POSIX_SPAWN_SETSIGDEF	874, 889, 897	posix_trace_get_filter()	947
POSIX_SPAWN_SETSIGMASK	889, 899	posix_trace_get_status	945
POSIX_TRACE_ADD_EVENTSET	947	posix_trace_get_status()	949
POSIX_TRACE_ALL_EVENTS	941	POSIX_TRACE_INHERITED	913
POSIX_TRACE_APPEND	914, 934	POSIX_TRACE_LOOP	73, 913-914, 934
posix_trace_attr_destroy()	909	POSIX_TRACE_NOT_FLUSHING	73
posix_trace_attr_getclockres()	911	POSIX_TRACE_NOT_FULL	72-74
posix_trace_attr_getcreatetime()	911	POSIX_TRACE_NOT_FULL	928
posix_trace_attr_getgenversion()	911	POSIX_TRACE_NOT_TRUNCATED	75, 951
posix_trace_attr_getinherited()	913	POSIX_TRACE_NO_OVERRUN	73-74, 945
posix_trace_attr_getlogfullpolicy()	913	posix_trace_open()	930, 953
posix_trace_attr_getlogsize()	916	POSIX_TRACE_OVERFLOW trace event	77
posix_trace_attr_getmaxdatasize()	916	POSIX_TRACE_OVERRUN	73-74
posix_trace_attr_getmaxsystemeventsz()	916	POSIX_TRACE_RESUME trace event	77
posix_trace_attr_getmaxusereventsz()	916	posix_trace_rewind()	930, 953
posix_trace_attr_getname()	911, 919	POSIX_TRACE_RUNNING	72, 956
posix_trace_attr_getstreamfullpolicy()	913, 920	POSIX_TRACE_SET_EVENTSET	947
posix_trace_attr_getstreamsize()	916, 921	posix_trace_set_filter()	947, 954
posix_trace_attr_init()	909, 922	posix_trace_shutdown()	932, 955
posix_trace_attr_setinherited()	913, 923	POSIX_TRACE_START trace event	77, 956
posix_trace_attr_setlogfullpolicy()	913, 923	posix_trace_start()	956
posix_trace_attr_setlogsize()	916, 924	posix_trace_status_info structure()	72
posix_trace_attr_setmaxdatasize()	916, 924	posix_trace_stop()	956
posix_trace_attr_setname()	911, 925	POSIX_TRACE_STOP trace event	77, 956
posix_trace_attr_setstreamfullpolicy()	913, 926	POSIX_TRACE_SUB_EVENTSET	947
posix_trace_attr_setstreamsize()	916, 927	POSIX_TRACE_SUSPENDED	72-73, 956
posix_trace_clear()	928	POSIX_TRACE_SYSTEM_EVENTS	941
posix_trace_close()	930	posix_trace_timedgetnext_event()	950, 958
POSIX_TRACE_CLOSE_FOR_CHILD	913	posix_trace_trid_eventid_open()	938, 959
posix_trace_create()	932	POSIX_TRACE_TRUNCATED_READ	75, 951
posix_trace_create_withlog()	932	POSIX_TRACE_TRUNCATED_RECORD	75, 951
POSIX_TRACE_ERROR trace event	77	posix_trace_trygetnext_event()	950, 960
posix_trace_event()	936	POSIX_TRACE_UNTIL_FULL	73, 913-914, 934
posix_trace_eventid_equal()	938	POSIX_TRACE_USER_EVENT_MAX	936
posix_trace_eventid_get_name()	938	POSIX_TRACE_WOPID_EVENTS	941
posix_trace_eventid_open()	936, 940	POSIX_TYPED_MEM_ALLOCATE	771-772
posix_trace_eventset_add()	941	867, 961, 963
posix_trace_eventset_del()	941	POSIX_TYPED_MEM_ALLOCATE_CONTIG
posix_trace_eventset_empty()	941	771-772, 867, 961, 963
posix_trace_eventset_fill()	941	posix_typed_mem_get_info()	961
posix_trace_eventset_ismember()	941	POSIX_TYPED_MEM_MAP_ALLOCATABLE
posix_trace_eventtypelist_getnext_id()	943	817, 963
posix_trace_eventtypelist_rewind()	943	posix_typed_mem_open()	963
posix_trace_event_info structure()	74	pow()	966
POSIX_TRACE_FILTER trace event	77, 947	powf()	966

<code>powl()</code>	966	<code>pthread_attr_getstackaddr()</code>	995
<code>pread()</code>	1167, 969	<code>pthread_attr_getstacksize()</code>	997
predefined stream		<code>pthread_attr_init()</code>	978, 998
standard error	37	<code>pthread_attr_setdetachstate()</code>	981, 999
standard input	37	<code>pthread_attr_setguardsize()</code>	983, 1000
standard output	37	<code>pthread_attr_setinheritsched()</code>	985, 1001
preempted thread	1034	<code>pthread_attr_setschedparam()</code>	987, 1002
<code>PRI</code>	18	<code>pthread_attr_setschedpolicy()</code>	989, 1003
<code>printf()</code>	402, 970	<code>pthread_attr_setscope()</code>	991, 1004
priority	39	<code>pthread_attr_setstack()</code>	993, 1005
<code>PRIO</code>	18	<code>pthread_attr_setstackaddr()</code>	995, 1006
<code>PRIO_INHERIT</code>	1084	<code>pthread_attr_setstacksize()</code>	997, 1007
<code>PRIO_PGRP</code>	540	<code>pthread_barrierattr_destroy()</code>	1012
<code>PRIO_PROCESS</code>	540	<code>pthread_barrierattr_getpshared()</code>	1014
<code>PRIO_USER</code>	540	<code>pthread_barrierattr_init()</code>	1012, 1016
process		<code>pthread_barrierattr_setpshared()</code>	1014, 1017
concurrent execution	394	<code>pthread_barrier_destroy()</code>	1008
setting real and effective user IDs	1307	<code>pthread_barrier_init()</code>	1008
single-threaded	394	<code>PTHREAD_BARRIER_SERIAL_THREAD</code>	1010
process creation	394	<code>pthread_barrier_wait()</code>	1010
process group		<code>pthread_cancel()</code>	1018
orphaned	308	<code>PTHREAD_CANCELED</code>	57, 1053
process group ID	536, 1299, 1311	<code>PTHREAD_CANCEL_ASYNCHRONOUS</code>	54
process ID, 1	308	1128
process lifetime	667	<code>PTHREAD_CANCEL_DEFERRED</code>	54, 1032, 1128
process scheduling	44	<code>PTHREAD_CANCEL_DISABLE</code>	54, 1128
process shared memory	1088	<code>PTHREAD_CANCEL_ENABLE</code>	54, 1128
process synchronization	1088	<code>pthread_cleanup_pop()</code>	1020
process termination	307	<code>pthread_cleanup_push()</code>	1020
<code>PROT</code>	16, 18	<code>pthread_condattr_destroy()</code>	1037
<code>PROT_EXEC</code>	771, 780	<code>pthread_condattr_getclock()</code>	1039
<code>PROT_NONE</code>	44, 771, 780	<code>pthread_condattr_getpshared()</code>	1041
<code>PROT_READ</code>	771, 780	<code>pthread_condattr_init()</code>	1037, 1043
<code>PROT_WRITE</code>	771, 773, 775, 780	<code>pthread_condattr_setclock()</code>	1039, 1044
<code>PS</code>	7	<code>pthread_condattr_setpshared()</code>	1041, 1045
<code>pselect()</code>	971	<code>pthread_cond_broadcast()</code>	1025
pseudo-random sequence generation functions	1164	<code>pthread_cond_destroy()</code>	1028
<code>PST8PDT</code>	1542	<code>pthread_cond_init()</code>	1028
<code>ps</code>	16	<code>PTHREAD_COND_INITIALIZER</code>	1028
<code>PTHREAD</code>	16	<code>pthread_cond_signal()</code>	1025, 1031
<code>pthread</code>	16	<code>pthread_cond_timedwait()</code>	1032
<code>pthread_atfork()</code>	976	<code>pthread_cond_wait()</code>	1032
<code>pthread_attr_destroy()</code>	978	<code>pthread_create()</code>	1046
<code>pthread_attr_getdetachstate()</code>	981	<code>PTHREAD_CREATE_DETACHED</code>	30, 981
<code>pthread_attr_getguardsize()</code>	983	<code>PTHREAD_CREATE_JOINABLE</code>	30, 981, 1063
<code>pthread_attr_getinheritsched()</code>	985	<code>PTHREAD_DESTRUCTOR_ITERATIONS</code>	1060
<code>pthread_attr_getschedparam()</code>	987	1065, 1468
<code>pthread_attr_getschedpolicy()</code>	989	<code>pthread_detach()</code>	1049
<code>pthread_attr_getscope()</code>	991	<code>pthread_equal()</code>	1051
<code>pthread_attr_getstack()</code>	993	<code>pthread_exit()</code>	1052
		<code>PTHREAD_EXPLICIT_SCHED</code>	985

<code>pthread_getconcurrency()</code>	1054	<code>pthread_rwlock_tryrdlock()</code>	1109, 1115
<code>pthread_getcpuclockid()</code>	1056	<code>pthread_rwlock_trywrlock()</code>	1116
<code>pthread_getschedparam()</code>	1057	<code>pthread_rwlock_unlock()</code>	1118
<code>pthread_getspecific()</code>	1060	<code>pthread_rwlock_wrlock()</code>	1116, 1120
<code>PTHREAD_INHERIT_SCHED</code>	985	<code>PTHREAD_SCOPE_PROCESS</code>	52-53, 991
<code>pthread_join()</code>	1062	<code>PTHREAD_SCOPE_SYSTEM</code>	52-53, 991
<code>PTHREAD_KEYS_MAX</code>	1065, 1468	<code>pthread_self()</code>	1127
<code>pthread_key_create()</code>	1065	<code>pthread_setcancelstate()</code>	1128
<code>pthread_key_delete()</code>	1069	<code>pthread_setcanceltype()</code>	1128
<code>pthread_kill()</code>	1071	<code>pthread_setconcurrency()</code>	1054, 1130
<code>pthread_mutexattr_destroy()</code>	1087	<code>pthread_setschedparam()</code>	1057, 1131
<code>pthread_mutexattr_getprioceiling()</code>	1092	<code>pthread_setschedprio()</code>	1132
<code>pthread_mutexattr_getprotocol()</code>	1094	<code>pthread_setspecific()</code>	1060, 1134
<code>pthread_mutexattr_getpshared()</code>	1096	<code>pthread_sigmask()</code>	1135
<code>pthread_mutexattr_gettype()</code>	1098	<code>pthread_spin_destroy()</code>	1137
<code>pthread_mutexattr_init()</code>	1087, 1100	<code>pthread_spin_init()</code>	1137
<code>pthread_mutexattr_setprioceiling()</code>	1092, 1101	<code>pthread_spin_lock()</code>	1139
<code>pthread_mutexattr_setprotocol()</code>	1094, 1102	<code>pthread_spin_trylock()</code>	1139
<code>pthread_mutexattr_setpshared()</code>	1096, 1103	<code>pthread_spin_unlock()</code>	1141
<code>pthread_mutexattr_settype()</code>	1098, 1104	<code>PTHREAD_STACK_MIN</code>	993, 995, 997, 1468
<code>PTHREAD_MUTEX_DEFAULT</code>	1080, 1098	<code>pthread_testcancel()</code>	1128, 1142
<code>pthread_mutex_destroy()</code>	1072	<code>PTHREAD_THREADS_MAX</code>	1046, 1468
<code>PTHREAD_MUTEX_ERRORCHECK</code>	1080, 1098	<code>ptsname()</code>	1143
<code>pthread_mutex_getprioceiling()</code>	1077	<code>putc()</code>	1144
<code>pthread_mutex_init()</code>	1072, 1079	<code>putchar()</code>	1146
<code>PTHREAD_MUTEX_INITIALIZER</code>	1072, 1079	<code>putchar_unlocked()</code>	485, 1147
<code>pthread_mutex_lock()</code>	1080	<code>putc_unlocked()</code>	485, 1145
<code>PTHREAD_MUTEX_NORMAL</code>	1080, 1098	<code>putenv()</code>	1148
<code>PTHREAD_MUTEX_RECURSIVE</code>	1080, 1098-1099	<code>putmsg()</code>	1150
<code>pthread_mutex_setprioceiling()</code>	1077, 1083	<code>putpmsg()</code>	1150
<code>pthread_mutex_timedlock()</code>	1084	<code>puts()</code>	1154
<code>pthread_mutex_trylock()</code>	1080, 1086	<code>pututxline()</code>	282, 1156
<code>pthread_mutex_unlock()</code>	1080, 1086	<code>putwc()</code>	1157
<code>pthread_once()</code>	1105	<code>putwchar()</code>	1158
<code>PTHREAD_ONCE_INIT</code>	1105	<code>pwrite()</code>	1649, 1159
<code>PTHREAD_PRIO_INHERIT</code>	1094	<code>pw_</code>	16
<code>PTHREAD_PRIO_NONE</code>	1094	<code>p_</code>	16
<code>PTHREAD_PRIO_PROTECT</code>	1081, 1094	<code>P_</code>	17
<code>PTHREAD_PROCESS_PRIVATE</code>	1014, 1041	<code>P_ALL</code>	1588
.....	1088, 1096, 1123, 1137	<code>P_PGID</code>	1588
<code>PTHREAD_PROCESS_SHARED</code>	1014, 1041	<code>P_PID</code>	1588
.....	1088, 1096, 1123, 1137	<code>qsort()</code>	1160
<code>pthread_rwlockattr_destroy()</code>	1121	<code>queue a signal to a process</code>	1364
<code>pthread_rwlockattr_getpshared()</code>	1123	<code>raise()</code>	1161
<code>pthread_rwlockattr_init()</code>	1121, 1125	<code>rand()</code>	1163
<code>pthread_rwlockattr_setpshared()</code>	1123, 1126	<code>random()</code>	605, 1166
<code>pthread_rwlock_destroy()</code>	1107	<code>RAND_MAX</code>	1163
<code>pthread_rwlock_init()</code>	1107	<code>rand_r()</code>	1163
<code>pthread_rwlock_rdlock()</code>	1109	<code>read from a file</code>	1170
<code>pthread_rwlock_timedrdlock()</code>	1111	<code>read()</code>	1167
<code>pthread_rwlock_timedwrlock()</code>	1113	<code>readdir()</code>	1174

readdir_r()	1174	remainderl()	1202
readlink()	1178	remove a directory	1219
readv()	1181	remove directory entries	1557
real user ID	97, 666	remove()	1204
realloc()	1183	remque()	607, 1206
realpath()	1185	remquo()	1207
REALTIME	104, 106-107, 109	remquof()	1207
	112-113, 115, 195, 201, 335	remquol()	1207
	683, 767, 769, 782-783, 785	rename a file	1211
	787, 790, 793, 795, 799, 820	rename()	1209
	1228-1232, 1235, 1243-1246	rewind()	1213
	1248, 1251, 1255, 1257, 1259	rewinddir()	1214
	1323, 1327, 1363, 1370, 1376	re_	16
	1514, 1517-1518	RE_DUP_MAX	1468
REALTIME THREADS	985, 989, 991	rindex()	1215
	1001, 1003-1004, 1057, 1077	rint()	1216
	1083, 1092, 1094, 1101-1102	RLIMIT_	18
	1131-1132	RLIMIT_AS	552
recv()	1187	RLIMIT_CORE	551
recvfrom()	1189	RLIMIT_CPU	551
recvmsg()	1192	RLIMIT_DATA	551
regcomp()	1195	RLIMIT_FSIZE	551
regerror()	1195	RLIMIT_NOFILE	551, 553
regexec()	1195	RLIMIT_STACK	551
regfree()	1195	rlim_	16
register fork handlers	976	RLIM_	18
REG_	18	RLIM_INFINITY	551-552
REG_constants		RLIM_SAVED_CUR	552
error return values of regcomp	1197	RLIM_SAVED_MAX	552
used in regcomp	1195	rmdir()	1218
REG_BADBR	1197	RMSGD	613
REG_BADPAT	1197	RMSGN	613
REG_BADRPT	1197	RNORM	613
REG_EBRACE	1197	round robin	46
REG_EBRACK	1197	round()	1221
REG_ECOLLATE	1197	roundf()	1221
REG_ECTYPE	1197	roundl()	1221
REG_EESCAPE	1197	routing	59
REG_EPAREN	1197	RPROTDAT	613
REG_ERANGE	1197	RPROTDIS	613
REG_ESPACE	1197	RPROTNORM	613
REG_ESUBREG	1197	RS	7
REG_EXTENDED	1195	RS_HIPRI	522, 612, 1150
REG_ICASE	1195	RTLD_	18
REG_NEWLINE	1195	RTLD_DEFAULT	259
REG_NOMATCH	1197	RTLD_GLOBAL	252, 256-257, 260
REG_NOSUB	1195	RTLD_LAZY	256, 259
REG_NOTBOL	1196	RTLD_LOCAL	257
REG_NOTEOL	1196	RTLD_NEXT	259-260
remainder()	1202	RTLD_NOW	256-257
remainderf()	1202	RTS	7

RTSIG_MAX.....	1468	SEEK_END.....	328, 440, 725
RUSAGE_.....	18	SEEK_GET.....	1213
RUSAGE_CHILDREN.....	554	SEEK_SET.....	42, 109, 115, 328, 440, 725
RUSAGE_SELF.....	554	SEGV.....	18
ru.....	16	select().....	971, 1242
s6.....	16	SEM.....	7
sa.....	16	semctl().....	1260
SA.....	18	semget().....	1263
SA_NOCLDSTOP.....	31, 1338-1339, 1342	semid.....	40
SA_NOCLDWAIT.....	305-306, 554, 1340, 1582	semop().....	1266
SA_NODEFER.....	1340	SEM.....	16
SA_ONSTACK.....	295, 1339	sem.....	16
SA_RESETHAND.....	147, 1339-1340	SEM.....	18
SA_RESTART.....	147, 974, 1339, 1353	sem_close().....	1243
SA_SIGINFO.....	1338-1339, 1342, 1363	sem_destroy().....	1244
scalb().....	1223	SEM_FAILED.....	1249
scalbln().....	1225	sem_getvalue().....	1245
scalblnf().....	1225	sem_init().....	1246
scalblnl().....	1225	SEM_NSEMS_MAX.....	1246, 1468
scalbn().....	1225	sem_open().....	1248
scalbnf().....	1225	sem_perm.....	40
scalbnl().....	1225	sem_post().....	1251
scanf().....	433, 1227	sem_timedwait().....	1253
schedule alarm.....	118	sem_trywait().....	1255
scheduling documentation.....	54	SEM_UNDO.....	1266
scheduling policy		sem_unlink().....	1257
round robin.....	46	SEM_VALUE_MAX.....	1246, 1248, 1468
SCHED_.....	16	sem_wait().....	1255, 1259
sched.....	16	send().....	1271
SCHED_FIFO.....	42, 45, 53, 296, 392	sendmsg().....	1273
.....	540, 829, 987, 989, 1057	sendto().....	1276
.....	1092, 1109, 1233, 1251	service name.....	424
sched_getparam().....	1229	session.....	308, 666, 1299, 1311
sched_getscheduler().....	1230	set cancelability state.....	1128
sched_get_priority_max().....	1228	set file creation mask.....	1548
sched_get_priority_min().....	1228	set process group ID for job control.....	1299
SCHED_OTHER.....	45, 48, 540, 989, 1057, 1233	set-group-ID.....	187, 302, 307, 332
SCHED_RR.....	42, 45-46, 53, 296, 392, 540	set-user-ID.....	302, 307, 492, 666
.....	829, 987, 989, 1057, 1109, 1233, 1251	SETALL.....	1260, 1263
sched_rr_get_interval().....	1231	setbuf().....	1279
sched_setparam().....	1232	setcontext().....	489, 1280
sched_setscheduler().....	1235	setgid().....	1281
SCHED_SPORADIC.....	42, 45, 47, 1109, 1233, 1251	setenv().....	1282
sched_yield().....	1238	seteuid().....	1284
SCM.....	18	setgid().....	1285
SCN.....	18	setgrent().....	270, 1287
SD.....	7	sethostent().....	272, 1288
security considerations....	189, 307, 666, 1299, 1400	setitimer().....	517, 1289
seed48().....	261, 1239	setjmp().....	1290
seekdir().....	1240	setkey().....	1292
SEEK_CUR.....	328, 440, 725	setlocale().....	1293

setlogmask()	208, 1297	SIGCLD	1342
setnetent()	274, 1298	SIGCONT	34, 306, 308, 665-666
setpgid()	1299	sigdelset()	1347
setpgrp()	1301	sigemptyset()	1348
setpriority()	540, 1302	SIGEV_	16
setprotoent()	276, 1303	sigev_	16
setpwent()	278, 1304	SIGEV_NONE	29, 42
setregid()	1305	SIGEV_SIGNAL	29, 1514
setreuid()	1307	SIGEV_THREAD	30
setrlimit()	551, 1309	sigfillset()	1350
setservent()	280, 1310	SIGFPE	1135, 1357
setsid()	1311	sighold()	1351
setsockopt()	1313	SIGHUP	203, 306, 308
setstate()	605, 1316	sigignore()	1351
setuid()	1317	SIGILL	1135, 1357
setutxent()	282, 1320	SIGINT	394, 1474
SETVAL	1260, 1263	siginterrupt()	1353
setvbuf()	1321	sigismember()	1355
shall	2	SIGKILL	666, 1338, 1342, 1351
shell	300, 308, 520, 536, 666, 1300, 1585	siglongjmp()	1356
job	666	signal generation and delivery	28
login	520	realtime	29
shell scripts		signal handler	1357
exec	300	signal()	1357
shell, login	300	signaling a condition	1026
SHM	7, 18	signals	28
shmat()	1329	signbit()	1359
shmctl()	1331	sigpause()	1351, 1360
shmdt()	1333	sigpending()	1361
shmget()	1334	SIGPIPE	325, 358, 414, 418, 441, 444, 1151, 1652
shmid	40	SIGPOLL	203, 611-612
SHMLBA	1329	sigprocmask()	1135, 1362
shm_	16	SIGPROF	517
SHM_	18	sigqueue()	1363
shm_open()	1323	SIGQUEUE_MAX	1363, 1468
shm_perm()	40	SIGQUIT	1474
SHM_RDONLY	1329	sigrelse()	1351, 1365
SHM_RND	1329	SIGRTMAX	29-30, 1363, 1370, 1374
shm_unlink()	1327	SIGRTMIN	29-30, 1363, 1370, 1374
should	2	SIGSEGV	44, 552, 817, 983, 1135, 1357
shutdown()	1336	sigset	1351, 1365
SHUT_	18	sigsetjmp()	1366
SIGABRT	92	SIGSTKSZ	1345
sigaction()	1338	SIGSTOP	29, 1338, 1342, 1351
sigaddset()	1344	sigsuspend()	1368
SIGALRM	118, 517, 1382, 1544, 1561	sigtimedwait()	1370
sigaltstack()	1345	SIGTSTP	29
SIGBUS	44, 773, 776, 1135	SIGTTIN	29, 362, 368, 1169
SIGCANCEL	1018	SIGTTOU	29, 325, 358, 414, 418
SIGCHLD	209, 305-306, 554, 579		441, 443, 1483, 1485, 1487, 1494
	1339, 1342, 1351, 1474, 1582, 1588		1497, 1652

SIGURG.....	612
SIGVTALRM.....	517
sigwait().....	1374
sigwaitinfo().....	1370, 1376
SIGXCPU.....	551
SIGXFSZ.....	551, 1535
SIG_.....	16, 18
SIG_BLOCK.....	1135
SIG_DFL.....	31, 295, 552, 1338, 1340, 1357
SIG_ERR.....	147, 1357
SIG_HOLD.....	1351
SIG_IGN.....	31, 295, 301, 305-306, 5541338, 1357, 1582
SIG_SETMASK.....	1135
SIG_UNBLOCK.....	1135
sin().....	1377
sin6_.....	16
sinf().....	1377
sinh().....	1379
sinhf().....	1379
sinhl().....	1379
sinl().....	1377, 1381
sin_.....	16
SIO.....	8
SIOCATMARK.....	1385
sival_.....	16
si_.....	16-17
SI_.....	16, 18
SI_ASYNCIO.....	32
SI_MSGQ.....	32
SI_QUEUE.....	32
SI_TIMER.....	32
SI_USER.....	32
sleep().....	1382
sl_.....	16
SND.....	18
SNDZERO.....	615
snprintf().....	402, 1384
SO.....	18
socketatmark().....	1385
socket I/O mode.....	60
socket out-of-band data.....	61
socket owner.....	60
socket queue limits.....	60
socket receive queue.....	61
socket types.....	59
socket().....	1387
socketpair().....	1389
sockets.....	58
address families.....	58
addressing.....	59
asynchronous errors.....	62
connection indication queue.....	62
Internet Protocols.....	66
IPv4.....	67
IPv6.....	67
local UNIX connections.....	66
options.....	63
pending error.....	60
protocols.....	59
signals.....	62
SOCK_.....	18
SOCK_DGRAM.....	66, 1387, 1389
SOCK_RAW.....	66
SOCK_SEQPACKET.....	66, 1387, 1389
SOCK_STREAM.....	66, 1387, 1389
SPI.....	8
SPN.....	8
sporadic server policy.....	
execution capacity.....	47
replenishment period.....	47
sprintf().....	402, 1391
spurious wakeup.....	1026
sqrt().....	1392
sqrtf().....	1392
sqrtl().....	1392
srand().....	1163, 1394
srand48().....	261, 1395
random().....	605, 1396
SS.....	8
sscanf().....	433, 1397
SSIZE_MAX.....	790, 806, 1167, 1178, 1419, 1649
ss_.....	16-17
SS_.....	18
SS_DISABLE.....	1345-1346
SS_ONSTACK.....	1345
stack size.....	978
stat().....	1398
statvfs().....	447, 1402
stderr.....	1403
STDERR_FILENO.....	1403
stdin.....	1403
STDIN_FILENO.....	859, 1403
stdio locking functions.....	373
stdio with explicit client locking.....	485
stdout.....	1403
STDOUT_FILENO.....	859, 1403
STR.....	18
strcasecmp().....	1405
strcat().....	1406
strchr().....	1407
strcmp().....	1408

strcoll()	1410	sun_	17
strcpy()	1412	superuser	97, 189, 682, 1557
strcspn()	1414	supplementary groups	189, 510
strdup()	1415	SVID	1366
STREAM	614, 616, 1150, 1168, 1651	SVR4	775, 820
stream		sv_	16
byte-oriented	37	SV_	18
wide-oriented	37	swab()	1459
STREAM head/tail	38	swapcontext()	729, 1460
stream-full-policy attribute	72-73, 76, 914	swprintf()	466, 1461
stream-min-size attribute	76, 917	swscanf()	475, 1462
streams	34	SWTCH	18
interaction with file descriptors	35	symbols	
stream orientation	37	POSIX.1	13
STREAMS	22, 203, 317, 336, 522	symlink()	1463
access	39	SYMLINK_MAX	397, 1463
multiplexed	618	SYMLOOP_MAX	146, 218, 318, 336, 389
overview	38	429, 448, 456, 462, 671, 759	
STREAM_MAX	340, 389, 859, 1468	827, 1185, 1275, 1278, 1468	
strerror()	1416	1536, 1565	
strerror_r()	1416	sync()	1465
strfmon()	1418	synchronously accept a signal	1371
strftime()	1422	sysconf()	1466
strlen()	1427	syslog()	208, 1473
strncasecmp()	1405, 1429	system crash	450
strncat()	1430	System III	189, 1550
strncmp()	1431	system interfaces	83
strncpy()	1432	system name	1550
strpbrk()	1433	system trace event type definitions	76
strptime()	1434	System V	118, 189, 303, 308, 330-331
strrchr()	1438	399, 536, 666, 753, 1219, 1311	
strspn()	1439	1342, 1366, 1489, 1550	
strstr()	1440	system()	1474
strtod()	1441	s_	16
strtof()	1441	S_	18
strtoimax()	1445	S_BANDURG	612
strtok()	1446	S_ERROR	611
strtok_r()	1446	S_HANGUP	612
strtol()	1449	S_HIPRI	611
strtold()	1441, 1451	S_IFBLK	758
strtoll()	1449, 1452	S_IFCHR	758
strtoul()	1453	S_IFDIR	758
strtoull()	1453	S_IFIFO	758
strtoumax()	1445, 1456	S_IFREG	758
strxfrm()	1457	S_INPUT	611
str_	16	S_IRGRP	321, 445, 758
st_	16	S_IROTH	321, 445, 758
ST_	18	S_IRUSR	321, 445, 758
ST_NOSUID	295, 447	S_IRWXG	758
ST_RDONLY	447	S_IRWXO	758
		S_IRWXU	758

S_ISGID.....	185-187, 758, 1535, 1650
S_ISUID.....	185-186, 758, 1535, 1650
S_ISVTX.....	185, 758, 1210, 1219, 1555
S_IWGRP.....	321, 445, 758
S_IWOTH.....	321, 445, 758
S_IWUSR.....	321, 445, 758
S_IXGRP.....	758
S_IXOTH.....	758
S_IXUSR.....	758
S_MSG.....	611
S_OUTPUT.....	611
S_RDBAND.....	611-612
S_RDNORM.....	611
S_WRBAND.....	611
S_WRNORM.....	611
TABSIZE.....	149, 723
tan().....	1478
tanf().....	1478
tanh().....	1480
tanhf().....	1480
tanh1().....	1480
tanl().....	1478, 1482
tcdrain().....	1483
tcflow().....	1485
tcflush().....	1487
tcgetattr().....	1489
tcgetpgrp().....	1491
tcgetsid().....	1493
TCIFLUSH.....	1487
TCIOFF.....	1485
TCIOFLUSH.....	1487
TCION.....	1485
TCOFLUSH.....	1487
TCOOFF.....	1485
TCOON.....	1485
TCP.....	18
TCSADRAIN.....	1496
TCSAFLUSH.....	1496
TCSANOW.....	1496
tcsendbreak().....	1494
tcsetattr().....	1496
tcsetpgrp().....	1499
TCT.....	8
tdelete().....	1501
TEF.....	8
telldir().....	1505
tempnam().....	1506
terminal access control.....	1489, 1497
terminal device name.....	1539
terminate a process.....	307
terminology.....	1
termios structure.....	1489
tfind().....	1501, 1508
tgamma().....	1509
tgammalf().....	1509
tgammal().....	1509
THR.....	8
thread cancelation.....	
cleanup handlers.....	57
thread creation.....	1047
thread creation attributes.....	978
thread ID.....	51, 1051
thread mutexes.....	51
thread scheduling.....	52
thread termination.....	1052
thread-safety.....	50, 373
thread-specific data key creation.....	1067
thread-specific data key deletion.....	1069
thread-specific data management.....	1061
threads.....	50
regular file operations.....	58
time().....	1511
timer ID.....	1516
timer.....	17
TIMER.....	17-18
TIMER_ABSTIME.....	49, 198, 1518
timer_create().....	1514
timer_delete().....	1517
timer_getoverrun().....	1518
timer_gettime().....	1518
TIMER_MAX.....	1468
timer_settime().....	1518
times().....	1521
timezone().....	1523
TMO.....	8
tmpfile().....	1524
tmpnam().....	1526
TMP_MAX.....	1506, 1525-1526
TMR.....	9
tms.....	16
tm.....	17
toascii().....	1528
tolower().....	1529
TOSTOP.....	325, 358, 414, 418, 441, 443, 1651
toupper().....	1530
towctrans().....	1531
towlower().....	1532
towupper().....	1533
TPI.....	9
TPP.....	9
TPS.....	9
trace event, POSIX_TRACE_ERROR.....	77

trace event, POSIX_TRACE_FILTER	77, 947	uname()	1550
trace event, POSIX_TRACE_OVERFLOW	77	undefined	2
trace event, POSIX_TRACE_RESUME	77	underlying function	36
trace event, POSIX_TRACE_START	77, 956	ungetc()	1552
trace event, POSIX_TRACE_STOP	77, 956	ungetwc()	1553
trace functions	79	unicast	67
trace-name attribute	76, 911	unlink()	1555
TRACE_EVENT_NAME_MAX	936, 938	unlockpt()	1559
TRACE_SYS_MAX	933	unsetenv()	1560
TRACE_USER_EVENT_MAX	936, 938	unspecified	2
TRACING	909, 911, 913, 916, 919-928	UP	10
.....	930, 932, 936, 938, 940-941	US-ASCII	624
.....	943-945, 947, 949-950, 953-956	user ID	
.....	958-960	real and effective	1307
TRAP_	18	setting real and effective	1307
TRC	9	user trace event type definitions	79
TRI	9	USER_PROCESS	282-283
TRL	9	usleep()	1561
trunc()	1534	UTC	1542
truncate()	1535	utime()	1563
truncation-status attribute	936	utimes()	1565
truncf()	1534, 1537	utim_	17
truncl()	1534, 1537	uts_	17
TSA	10	ut_	17
tsearch()	1501, 1538	va_arg()	1567
TSF	10	va_copy()	1567
TSH	10	va_end()	1567
TSP	10	va_start()	1567
TSS	10	VDISCARD	18
ttynam()	1539	VDSUSP	18
ttynam_r()	1539	Version 7	118, 189, 666, 1550
TTY_NAME_MAX	1468, 1539	vfork()	1568
tv_	16-17	vfprintf()	1570
twalk()	1501, 1541	vfscanf()	1571
TYM	10	vfwprintf()	1572
tzname	1542	vfwscanf()	1573
TZNAME_MAX	1468	VISIT	1501, 1541
tzset()	1542, 1542	VLNEXT	18
t_uscalar_t	613	vprintf()	1570, 1574
ualarm()	1544	VREPRINT	18
uc_	16-17	vscanf()	1571, 1575
UINT	18	vsprintf()	1570, 1576
UINT_MAX	118, 1383	vsprintf()	1570, 1576
UIO_MAXIOV	17	vsscanf()	1571, 1577
ulimit()	1546	VSTATUS	18
ULLONG_MAX	1454	vswprintf()	1572, 1578
ULONG_MAX	1454, 1625	vswscanf()	1573, 1579
UL_	17	VWERASE	18
UL_GETFSIZE	1546	vwprintf()	1572, 1580
UL_SETFSIZE	1546	vwscanf()	1573, 1581
umask()	1548	wait for process termination	1585

wait for thread termination.....	1063	WIFCONTINUED.....	1583
wait()	1582	WIFEXITED.....	1583
waitid().....	1588	WIFSIGNALED.....	1583
waiting on a condition.....	1033	WIFSTOPPED.....	1583, 1586
waitpid().....	1582, 1590	wmemchr().....	1639
WARNING.....	384	wmemcmp().....	1640
warning		wmemcpy().....	1641
OB.....	6	wmemmove().....	1642
OF.....	6	wmemset().....	1643
WCONTINUED.....	1582, 1588	WNOHANG.....	1342, 1582, 1588
wcrtomb().....	1591	WNOWAIT.....	1588
wcscat().....	1593	wordexp().....	1644
wcschr().....	1594	wordfree().....	1644
wcscmp().....	1595	wprintf().....	466, 1648
wscoll().....	1596	WRDE.....	18
wscpy().....	1597	WRDE_APPEND.....	1645
wcscspn().....	1598	WRDE_BADCHAR.....	1646
wcsftime().....	1599	WRDE_BADVAL.....	1646
wcslen().....	1601	WRDE_CMDSUB.....	1646
wcsncat().....	1602	WRDE_DOOFFS.....	1645
wcsncmp().....	1603	WRDE_NOCMD.....	1645
wcsncpy().....	1604	WRDE_NOSPACE.....	1646
wcspbrk().....	1605	WRDE_REUSE.....	1645
wcsrchr().....	1606	WRDE_SHOWERR.....	1645
wcsrtombs().....	1607	WRDE_SYNTAX.....	1646
wcsspn().....	1609	WRDE_UNDEF.....	1645
wcsstr().....	1610	write to a file.....	1653
wcstod().....	1611	write().....	1649
wcstof().....	1611	writew().....	1657
wcstoimax().....	1614	wscanf().....	475, 1659
wcstok().....	1615	WSTOPPED.....	1588
wcstol().....	1617	WSTOPSIG.....	1583
wcstold().....	1611, 1620	WTERMSIG.....	1583
wcstoll().....	1617, 1621	WUNTRACED.....	1582, 1585
wcstombs().....	1622	XSI.....	10
wcstoul().....	1624	XSI interprocess communication.....	39
wcstoumax().....	1614, 1627	XSR.....	11
wcswcs().....	1628	X_OK.....	97
wcswidth().....	1629	y0().....	1660
wcsxfrm().....	1630	y1().....	1660
wctob().....	1632	yn().....	1660
wctomb().....	1633	zombie process.....	305
wctrans().....	1635		
wctype().....	1636		
wcwidth().....	1638		
WEOF.....	82, 647-648, 650-651, 653-660		
.....	1532-1533, 1553		
WEXITED.....	1588		
WEXITSTATUS.....	1583		
we_.....	17		
wide-oriented stream.....	37		